View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at VSB Technical University of Ostrava

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Návrhové vzory v objektově orientovaném a funkcionálním programování a jejich integrace

# Comparision and Integration of Design Patterns Implemented in Functional and Object Oriented Languages

2015

Martin Maděra

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Martin Maděra**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Návrhové vzory v objektově orientovaném a funkcionálním programování a jejich integrace
Comparision and Integration of Design Paterns Implemented in Functional and Object Oriented Languages

Zásady pro vypracování:

Diplomant provede srovnání obecného znovupoužitelného řešení běžně se objevujících problémů (návrhových vzorů) v objektově orientovaném programovacím jazyku a ve funkcionálních jazycích. Dále se bude věnovat možnostem integrace částí systému implementovaných v objektově orientovaném a funkcionálním programovacím jazyku a vypracuje ukázkovou aplikaci, na které bude možno toto srovnání pozorovat.
1. Proveďte srovnání návrhových vzorů implementovaných ve vybraném funkcionálním jazyku a v objektově orientovaných jazyku.
2. Vypracujte vzorovou aplikaci demonstrující použití v obou typech jazyků.
3. Proveďte integraci vybraných částí např. uživatelského rozhraní v objektově orientovaném jazyce a funkčnosti ve funkcionálním jazyce.

Seznam doporučené odborné literatury:

[1] Chris Okasaki: Purely Functional Data Structures, ISBN-10: 0521663504, Cambridge University Press (June 13, 1999).
[2] Richard Bird: Pearls of Functional Algorithm Design, ISBN-10: 0521513383, Cambridge University Press; 1 edition (November 1, 2010).
[3] Simon Marlow: Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming, ISBN-10: 1449335942, O'Reilly Media; 1 edition (August 18, 2013).
Další literatura dle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Svatopluk Štolfa, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

## Prohlášení studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Martin Maděra

# Abstrakt

V této práci jsou uvedeny vybrané kreacionální a behaviorální návrhové vzory, které mají význam i ve funkcionálním programování. Strukturální vzory uvedeny nejsou, neboť jsou příliš spjaty s objektově orientovaným programováním a situace, které řeší, se ve funkcionálním programování zpravidla nevyskytují. Popsány jsou také některé vzory typické pro funkcionální programování a také způsob, jak některé funkcionální jazyky přistupují ke zpracování paralelních výpočtů. Dále jsou popsány možnosti integrace modulů vytvořených ve funkcionálním a objektově orientovaném programovacím jazyku a tyto možnosti jsou demonstrovány na praktickém příkladu.



# Klíčová slova

Návrhové vzory, funkcionální programování, Scala, Clojure

# Abstract

This thesis describes selected creational and behavioral design patters which are significant even for the functional programming. The structural patterns are omitted because they are closely related to the object oriented programming. Situations which are addressed by the structural patterns usually do not occur in the functional programming. Some patterns typical for the functional programming are also described along with ways how parallelism is implemented in some functional programming languages. Last section of the thesis describes ways to integrate modules created in both functional and object oriented programming languages and these ways are demonstrated on a practical example.



# Key Words

Design Patterns, Functional Programming, Scala, Clojure

# Obsah

# 1. Úvod

V posledních letech je patrná změna trendu ve vývoji nových procesorů, kdy se již nezvyšuje jejich výkon zvýšením pracovní frekvence, ale zvýšením počtu jejich jader [1]. Tato situace přináší trend zvýšeného zájmu o paralelní programování, což je patrno i z návrhu některých mainstreamových programovacích jazyků a knihoven. Například zatímco jazyky Java a C# potažmo jejich základní knihovny mají základní podporu pro paralelizaci již od svých prvních verzí (viz rozhraní Runnable v JDK1.0 [2] nebo třída Thread v .Net Frameworku 1.1 [3]), teprve s postupem času začaly přibývat sofistikovanější nástroje (např. Task Parallel Library [4] nebo Parallel LINQ [5] v Microsoft .Net Frameworku 4 nebo Parallel Streams v Javě 8 [6]). V roce 2012 dokonce do jazyka C# přibyly syntaktické konstrukce pro paralelní programování [7].

Nicméně i tyto příjemné vymoženosti ne vždy řeší typické problémy, jako je deadlock nebo race condition. Toto je důsledek povahy imperativního programování, které předpokládá možnost téměř kdykoliv změnit jakoukoliv hodnotu. Funkcionální programovací jazyky Clojure a Scala, představené v této práci, umožňují vyjádřit neměnnost stavu a referenční transparentnost, díky čemuž mohou poskytnout nástroje, jak se zmíněným problémům vyhnout. Z tohoto důvodu se lépe hodí na řešení úloh, kde je třeba paralelně zpracovávat data. [8]

Zatímco imperativní programovací jazyky se v posledních letech staly de facto hlavním proudem v programování a existuje mnoho publikací popisující dobré praktiky a styly práce, zdrojů zaměřených na funkcionální programování je ve srovnání s nimi málo. Tato práce si klade za cíl ukázat, jak by ve funkcionálních programovacích jazycích vypadala řešení úloh, které pokrývají základní návrhové vzory (viz [9]), a jak lze integrovat části systémů vytvořených v imperativních a funkcionálních programovacích jazycích.

# 2. Návrhové vzory

Návrhový vzor lze definovat jako příkladné řešení určité úlohy, která se vyskytuje stále znovu [9]. V současné době jsou mnohými odborníky považovány za důležitou znalost a jsou na ně často mířeny otázky při přijímacích pohovorech na programátorské pracovní pozice. Problém je v tom, že návrhové vzory publikované v knize [9] (lidově známá též jako Gang of Four Design Patterns nebo GoF DP), přináší řešení úloh pouze v imperativních jazycích – C++ a Smalltalku (tedy příklad staticky typovaného objektově orientovaného jazyka a dynamicky typovaného). Tato kapitola má za cíl přenést vybrané návrhové vzory do prostředí funkcionálního programování.

## 1.1. Jsou stále potřeba i ve funkcionálním programování?

Stále v programátorské praxi řešíme stejné typy úloh – řídíme přístup (vzor Proxy), vytváříme složité struktury (vzor Builder), sledujeme stav jiného objektu (vzor Observer)… Nicméně nelze zapomínat, že návrhové vzory v [9] byly definovány v kontextu především jazyka C++.
V některých pozdějších jazycích některé tyto vzory mohou zmizet, jako např. v jazyce C#, který podporuje vytvoření události (event) v jednom objektu a v druhém objektu reakci, pokud daná událost nastane (více informací viz [10]). Tím pádem z programů napsaných v jazyce C# zmizela drtivá většina případů pro použití vzoru Observer (definice vzoru Observer viz [9], strana 293).

Už v roce 1997, 4 roky po publikaci knihy Design Patterns, ve své přednášce Peter Norvig zmínil, že v dynamických jazycích Dylan a Lisp „*16 z 23 návrhových vzorů* [publikovaných v [9]] *je buď neviditelných nebo jednodušších*“ [11] proto, že tyto jazyky mají méně omezení a více konstrukcí. Pojmem *neviditelný* myslí Petr Norvig takový vzor, který je přirozeným konstruktem jazyka a nikoliv uměle vytvořeným vzorem, např. to, že funkce je tzv. prvotřídní (frist-class) typ v daném jazyku. Tím, že můžeme jako argument jedné funkce předat další funkci, může „zmizet“ použití vzorů Command, Strategy, Template-Method a Visitor (viz. [11] strana 10).[1]

Jazyky použité v prezentaci Petra Norviga jsou především objektově orientované imperativní jazyky, stejně jako C++ a Smalltalk použité v knize Ericha Gammy a spol. Ve funkcionálním programování se některé typy úloh řeší jiným způsobem, nebo neexistují vůbec. Jiné jsou však stejné a návrhové vzory tak, jak je popsali Gamma a spol. zůstanou nezměněny. V této práci budou uvedeny některé kreacionální a behaviorální návrhové vzory z jejich knihy, které mají svůj smysl i ve funkcionálním programování. Naopak strukturální vzory budou vynechány, neboť jsou velmi spjaté s objektově orientovaným programováním a často řeší situace, které ve funkcionálním programování nenastávají nebo v případě jazyků kombinujících prvky z objektově orientovaného i funkcionálního programování se řeší shodně.

[1] Za zmínku stojí, že v jazyce C++ lze ve standardu C++11 (r. 2011) pracovat s lamda výrazy. Ty ale v jazyce C++ v době psaní knihy [9] nebyly dostupné (tehdy ani jazyk C++ nebyl ANSI standardem, to přišlo až v roce 1998) a proto nejsou v knize zmíněny.
Ukazatele na funkci a reference funkcí se samozřejmě dají využít také, ale taková funkce musí být předem definována – nelze jimi vytvořit anonymní funkci „jen tak“ a referenci na ni přiřadit do proměnné, jako v případě lambda výrazů.

## 1.2. Programovací jazyky Scala a Clojure

Pro příklady v této práci budou použity jazyky Scala a Clojure. Jedná se o jazyky sloužící pro tvorbu programů běžících nad Java Virtual Machine a programy v nich napsané lze jednoduše integrovat s dalšími programy a knihovnami běžících nad JVM (např. napsaných v Javě nebo Groovy).

Scala[2] je staticky typovaný hybridní objektově-orientovaný a funkcionální programovací jazyk se syntaxí velmi podobnou Javě. Zajímavé je, že je možné v něm programovat podobným imperativním stylem jako např. v Javě (jen vynechat zbytečné středníky a závorky) nebo funkcionálním stylem podobně jako např. v jazyce Haskell.

Název Scala je akronymem k „scalable language“, jak tvrdí jeho hlavní autor [12]. Hlavním motem je *„Mějte to nejlepší z obou světů. Konstruujte elegantní hierarchie tříd pro maximální znovupoužitelnost kódu a rozšiřitelnost, implementujte jejich chování pomocí funkcí vyšších řádů. Nebo cokoliv mezi tím.“*[13] Tvůrci jazyka se snažili při jeho vytváření těchto mot držet a vytvořili jazyk s mnoha dobrými vlastnostmi obou světů. Díky těmto vlastnostem si získává celkem dost popularity mezi Java programátory.

Naproti tomu Clojure[3] je dynamicky typovaný funkcionální programovací jazyk z rodiny LISP-1 (1 jmenný prostor společně jak pro funkce, tak pro proměnné). Jeho asi nejzajímavější vlastností je Software Transaction Memory systém, díky kterému je možno měnit hodnoty proměnných v transakcích, což usnadňuje sdílení paměti při paralelním zpracování dat. V tomto jazyku a jeho základní knihovně je kladen velký důraz na vytváření programů pomocí skládání funkcí a oddělení stavu dat a chování funkcí od sebe (funkce by měly být referenčně transparentní, jejich výsledek by měl být v čase pro stejné vstupy stejný).

Jazyk Clojure je vnímán jako méně populární než Scala[4], nicméně dle mého názoru je zajímavější, především díky vlastnostem LISPu jako takového.

[2] Kompilátor a běhové prostředí Scaly je možno stáhnout z oficiálních stránek jazyka http://www.scala-lang.org/ .
[3] Kompilátor a běhové prostředí Clojure je možno stáhnout z oficiálních stránek jazyka http://clojure.org/.
[4] např. 16. 6. 2014 bylo na stackoverflow.com (jeden z nejznámějších serverů s odbornou diskusí k programování) možno nalézt 26 319 otázek týkajících se jazyku Scala a 7 568 otázek týkajících se programování v Clojure. V TIOBE Indexu pro červen 2014, který měří počet odkazů nalezených vyhledávači pro daný programovací jazyk, se jazyk Clojure nedostal do první 50, Scala se umístila na 41. místě mezi jazyky jako Ada, Haskell, Lua či TCL. [14]

## 1.3. Kreacionální vzory

Kreacionální (občas též vytvářecí) návrhové vzory slouží k abstrahování procesu vytvoření instance třídy [9]. Pomáhají vytvářet hierarchie objektů preferující kompozici nad dědičností a zapouzdřit detaily o tom, jak přesně jsou takové objekty vytvářeny a komponovány.

Gamma a spol. definují v [9] celkem 5 kreacionálních vzorů – abstract factory, builder, factory method, prototype a singleton. Z těchto byly vybrány abstract factory, builder a singleton, které mají své místo i v jazycích Scala a Clojure a jejich implementace využívají vlastností těchto jazyků – nejsou to holé přepisy vzoru z Javy do jiného jazyka. Vzory factory method a prototype by byly příliš podobné implementacím v jazyku Java, neboť v nich lze hůře využít vlastností funkcionálního programování.

### 1.3.1. Singleton

Singleton je nejjednodušší z návrhových vzorů. Pomocí tohoto návrhového vzoru lze zajistit, že instance dané třídy bude vždy pouze 1 a ta bude v aplikaci globálně přístupná. [9]

V jazyku Java si tuto vlastnost si zajišťuje sama třída a to tak, že má privátní konstruktor (její instanci tedy nelze vytvořit jinak, než z jí samé) a k instanci se přistupuje pomocí statické getter metody. V této getter metodě se zkontroluje, zdali již není instance třídy vytvořena (uchovává se v statické proměnné), případně se vytvoří. Takto získanou instanci je pak možno běžně používat, viz následující ukázky:

```
01 public class SingletonClass {
02     private static SingletonClass _instance;
03
04     private SingletonClass() {}
05
06     public static SingletonClass getInstance() {
07         if (null == _instance) {
08             _instance = new SingletonClass();
09         }
10         return _instance;
11     }
12
13     public int computeSum(int a, int b) {
14         return a + b;
15     }
16 }
```

*Ukázka kódu 1 Singleton třída v jazyku Java*

```
01    int sum = SingletonClass.getInstance().computeSum(5, 3);
```

*Ukázka kódu 2 Použití Singleton třídy v jazyku Java*

Jazyk Scala umožňuje kromě tříd vytvořit i tzv. Object. Takový objekt je automaticky singleton.

```
01    object Singleton {
02      def computeSum(a:Int, b:Int) = a + b
03    }
```
*Ukázka kódu 3 Objekt singleton v jazyku Scala*

Řádek 1 v předchozí ukázce definuje Singleton jako objekt. Na řádku 2 je vidět definice metody computeSum, která přebírá 2 parametry typu integer (zde třída scala.Int). Tělo metody se skládá z jediného výrazu, a + b. U této metody není uveden návratový typ – díky typové inferenci překladač ví, že návratový typ bude celé číslo.

Zavolání metody je také jednodušší:

```
01    object Main extends App {
02      val sum = Singleton.computeSum(5, 3)
03    }
```
*Ukázka kódu 4 Volání metody singleton objektu v jazyku Scala*

V ukázce 4 stojí za zmínku řádek 1 – hlavní třída aplikace je ve Scale také objekt, rozšiřující App. Následující řádek definuje *sum* jako *hodnotu*, nikoliv proměnnou – její hodnotu tedy nelze dále v programu změnit.

V jazyku Scala tedy máme kratší kód, ve kterém je méně „rušivých“ znaků (závorky, středníky…), ale přesto obsahuje více informací, než kód v Javě. Toho lze využít při verifikaci kódu a při optimalizacích během kompilace.

Zajímavější příklad je v jazyku Clojure. Jedná se o jazyk, který není primárně objektový (resp. podporuje všechny vlastnosti objektově orientovaného programování, ale především pro spolupráci s knihovnami napsanými v jazyku Java). Samozřejmě tedy lze vytvořit třídu a implementovat vzor Singleton tak, jak jej popsali Gamma a spol v [9], ale je to zbytečně pracné a nepřinese to žádnou výhodu.

```
01    (ns singleton)
02
03    (def state (ref {})) ;reference na stav
04
05    (defn computeSum [a b]
06      (+ a b))
```
*Ukázka kódu 5 Singleton v jazyku Clojure*

```
01    (singleton/computeSum 5 6)
```
*Ukázka kódu 6 Volání metody singleton objektu v jazyku Clojure*

V ukázce kódu 5 je vidět, že není vytvářena třída – na řádku 1 je pouze definice jmenného prostoru (z něj však může kompilátor vygenerovat třídu, je-li to vhodné pro integraci s Java projektem). Jelikož se zde nevytváří třída, je třeba „stav objektu“ držet v proměnné, nejčastěji se k tomuto účelu používá hash mapa (pokud je tedy vůbec potřeba stav objektu udržovat). Ta je definována na řádku 3. Jedná se o proměnnou typu Ref[5], typ umožňující držet změnitelný stav pomocí Software

[5] Více informací o tomto typu je možné nalézt na oficiální stránce jazyka http://clojure.org/refs

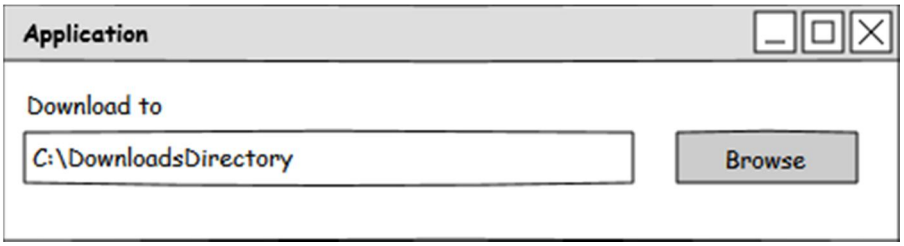
transactional memory. Přístup k Ref probíhá v transakci – pokud se tedy jeho hodnota změní během transakce (např. v transakci z jiného vlákna), transakce selže a je opakována. Hodnota může být přečtena i mimo transakci, ale při pokusu o zápis mimo transakci se vyvolá výjimka. Tím je zaručena bezpečnost při práci s vlákny, což v předchozích ukázkách není.

Ukázka 6 ukazuje zavolání funkce *computeSum*, a to při použití názvu jmenného prostoru a funkce. Samozřejmě do aktuálního jmenného prostoru lze pomocí makra :use vložit definice z jmenného prostoru *singleton*, ale v tomto případě by to bylo spíše kontraproduktivní.

#### *1.3.1.1 Praktický příklad*

Přestože *singleton* je nejjednodušší návrhový vzor, jeho použití v reálných programech je omezené spíše na řízení a abstrakci přístupu ke zdrojům sdíleným přes celou aplikaci, ke kterým vždy chceme přistupovat vždy přes 1 bod. Je možné jej využít například v přístupu k uživatelským nastavením aplikace nebo (v určitých případech) při logování. V této podkapitole bude uveden realistický příklad návrhového vzoru singleton, použitého pro přístup k uživatelským nastavením aplikace.

Například máme-li aplikaci, která dává uživateli na výběr adresář, kam stáhnout soubor ze sítě, je dobrým zvykem uživateli předvyplnit cestu, kterou zadal minule, neboť se dá očekávat, že i následující soubor bude chtít stáhnout do stejného místa. Takovéto informace lze uložit např. v konfiguraci programu specifické pro uživatele.




*Obrázek 1 Okno fiktivní aplikace pro příklad vzoru singleton*

Při otevření okna aplikace lze načíst název adresáře, do kterého uživatel posledně uložil soubor. Při zavření lze zkontrolovat, zda-li byl současný soubor uložen do stejného či jiného adresáře a konfiguraci změnit.

```
01    public class Config {
02        private static Config _instance;
03        private HashMap<String, String> _conf = new HashMap<String,
String>();
04
05        private static String DefaultPathKey = "DefaultPath";
06
07        private Config() {
08            //read the configuration from the disk and populate conf
hashmap
09        }
10
11        public static Config getInstance() {
```

```
12              if (_instance == null) {
13                  _instance = new Config();
14              }
15
16              return _instance;
17          }
18
19          public String getDefaultPath() {
20              return _conf.get(DefaultPathKey);
21          }
22
23          public void setDefaultPath(String newPath) {
24              _conf.put(DefaultPathKey, newPath);
25              //save the configuration to disk
26          }
27      }
```

*Ukázka kódu 7 Příklad singleton třídy v jazyku Java*

V třídě *Config* v ukázce kódu 7 lze vidět příklad singleton třídy, která (při vytvoření instance) načte uživatelské nastavení a po změně nastavení je opět uloží na disk. Kód pro samotné načtení a uložení na disk je vynechán, protože není pro tento vzor důležité, navíc možností realizace takové funkčnosti je mnoho. Samotné uživatelské nastavení je po celou dobu udržováno v paměti (HashMap) a pro jednoduchost se předpokládá, že se vždy budou ukládat řetězcová data. Je dobrou praxí řetězcové klíče uchovávat jako konstanty pro jednoduchost přejmenování (řádek 05 v předchozí ukázce), přestože některá moderní vývojová prostředí dokáží refaktorovat i řetězce přímo v kódu.

```
01    public class Window extends JFrame implements WindowListener {
02        private JTextField _downPathField;
03
04        public Window() {
05            _downPathField = new JTextField();
06
_downPathField.setText(Config.getInstance().getDefaultPath());
07        }
08
09
10        @Override
11        public void windowClosed(WindowEvent e) {
12            String newPath = _downPathField.getText();
13            if (!
newPath.equals(Config.getInstance().getDefaultPath())) {
14                Config.getInstance().setDefaultPath(newPath);
15            }
16        }
17    //ohter WindowListener overrides
18    }
```

*Ukázka kódu 8 Použití singleton třídy z jazyku Java*

V ukázce kódu 8 lze vidět použití třídy *Config*. Jedná se o SWING okno podobné jako na obrázku 1. Textové pole, které zobrazuje cestu, kam uložit soubor, je v proměnné *_downPathField*. V konstruktoru okna je tato proměnná nastavena a je nastaven i výchozí text tohoto pole na hodnotu získanou z nastavení pomocí třídy *Config*. Při zavření okna (metoda *windowClosed*) je zkontrolována cesta – obsah textového pole *_downPathField* – a je-li jiná než výchozí, je uložena do uživatelských nastavení. Z ukázky je záměrně vynechán zbytek metod rozhraní *WindowListener* a inicializace ovládacích prvků, neboť pro tuto ukázku nejsou důležité.

```
01    object Config {
02
03      private val _conf: mutable.HashMap[String, String] = new
mutable.HashMap[String, String]
04      private val DefaultPathKey: String = "DefaultPath"
05
06
07      def getDefaultPath: String =
08        _conf.get(DefaultPathKey).getOrElse("")
09
10      def setDefaultPath(newPath: String) {
11        _conf.put(DefaultPathKey, newPath)
12        //save the settings to disk
13      }
14
15      {
16        //load the settings from disk
17      }
18
19    }
```

*Ukázka kódu 9 Příklad singleton objektu v jazyku Scala*

V ukázce kódu 9 lze vidět kód pro singleton objekt, podobný ukázce 7. Definuje-li se *Config* jako *object* spíše než *class*, automaticky je tím zajištěno, že bude existovat vždy „jediná správná“ instance tohoto objektu. Objekt je v jazyku Scala vždy singleton, proto zde není potřeba mít metodu *getInstance*. Stejně tak objekty v jazyku Scala nemají konstruktory – veškerý kód těla je evaluován. Kód, který by v jazyku Java byl v konstruktoru, je nyní v anonymním bloku začínajícím na řádku 15 . Za zmínku také stojí, že *_conf* a *DefaultPathKey* jsou definovány jako *val*, tedy nikoliv proměnné, ale neměnné hodnoty. Pro tento program to znamená, že nikde v programu nebudou změněny a budou vždy referencovat stejný objekt v paměti. Další rozdíl lze vidět na řádku 08 . V této ukázce není pro držení hodnot v paměti použita třída *java.util.HashMap*, ale *scala.collection.mutable.HashMap*. Přestože lze použít i první jmenovanou, v jazyku Scala je výhodnější použít kolekce ze základní knihovny tohoto jazyka a při komunikaci s moduly v jazyku Java je převést na Java typy. Metoda *get* v tomto případě nevrátí objekt daného typu (v tomto případě objekt typu *String* nebo null), ale typ *Option[String]*. Jedná se o kontejner, který v případě, že je požadovaná hodnota nalezena, bude obsahovat referenci na nalezený objekt a v případě, že hodnota nebude nalezena, bude prázdný. Lze se tak vyhnout časté chybě, kdy programátor zapomene ověřit, zda-li se skutečně vrátila reference na instanci, či null. Nyní je tato kontrola vynucena. Funkce *getOrElse* typu *Option* vrátí buď nalezenou hodnotu, nebo výchozí hodnotu předanou jako parametr funkce *getOrElse*.

```
01    class Window extends JFrame with WindowListener {
02      private val _downPathField: JTextField = new JTextField
03      _downPathField.setText(Config.getDefaultPath)
04
05      def windowClosed(e: WindowEvent) {
06        val newPath = _downPathField.getText
07        if (!(newPath == Config.getDefaultPath)) {
08          Config.setDefaultPath(newPath)
09        }
10      }
11
12      //Other WindowListener overrides
13    }
```

*Ukázka kódu 10 Použití singleton objektu v jazyku Scala*

Ukázka kódu 10 se velmi podobá kódu v ukázce pro jazyk Java. Hlavním rozdílem je především absence volání funkce *getInstance*, neboť *Config* je definován jako *object* a tedy automaticky existuje právě 1 instance tohoto objektu. Lze také poukázat na fakt, že prázdné závorky *()* lze v jazyku Scala vynechat a také to, že třída *Window* nemá viditelně definovaný konstruktor, viz výše.

```
01    (ns p-singleton-clojure.config)
02
03    (def -settings (atom {}))
04
05    (defn get-default-path
06      []
07      (:defPath @-settings))
08
09    (defn set-default-path
10      [newPath]
11      (swap! -settings assoc :defPath newPath)
12    ;; save settings)
13
14    ;; load the settings from disk and poulate -settings
```

*Ukázka kódu 11 Příklad singleton v jazyku Clojure*

V jazyku Clojure lze návrhový vzor singleton implementovat jako jmenný prostor, viz ukázka 11. Jelikož se opět celý kód evaluuje, není třeba definovat konstruktor – načtení nastavení proběhne při prvním přístupu k prvkům tohoto jmenného prostoru. Člen *–settings* je definován jako *atom*, jeden z referenčních typů, s kterým lze operovat synchronně a nekoordinovaně (operace pro změnu hodnoty je synchronní (blokující současné vlákno) a atomická, ale neprobíhá uvnitř transakce). Dereferencovat jej lze pomocí funkce *deref* nebo pomocí makra *@*. Fakt, že název – *settings* začíná pomlčkou, znamená, že jde o privátního člena daného jmenného prostoru. Typem se jedná o *HashMap* (je vytvořen jako prázdná HashMap – *{}* – viz řádek 03 ). Ve funkci *get-default-path* lze vidět ukázku přístupu k *HashMap*. Použije-li se klíč mapy (zde klíčové slovo – začíná znakem „:“ ) na místě funkce, vrácená hodnota je hodnota z mapy (je-li nalezena), případně *nil*, ekvivalent k *null*, pokud nalezena není. Makro *@* představuje dereferenci referenčního typu.

Funkce *swap!* ve funkci *set-default-path* představuje změnu hodnoty atomu, jedná se o synchronní a atomickou operaci. První parametr této funkce je samotný (nedereferencovaný) atom, jehož hodnota se má změnit, změnovou funkci, která vrací novou hodnotu a jako první parametr přebírá současnou hodnotu atomu, a dále další parametry změnové funkce.

```
01    (ns p-singleton-clojure.core
02      (:require [p-singleton-clojure.config :as config])
03      (:import [javax.swing JFrame JTextField]
04               [java.awt.event WindowListener]))
05
06    (defn create-win-closed-listener
07      [downPathField]
08      (proxy [WindowListener] []
09         (windowClosed [event]
10           (let [user-value (.getText downPathField)
11                 current-value (config/get-default-path)]
12             (if-not (= user-value current-value)
13               (config/set-default-path user-value))))))
14
15    (defn create-window
16      []
17      (let [window (JFrame.)
18            downPathField (JTextField.)
19            windowListener (create-win-closed-listener downPathField)]
20        (.setWindowListener window windowListener)
21        (.setText downPathField (config/get-default-path))
22        window))
```

*Ukázka kódu 12 Použití singleton v jazyku clojure*

V ukázce výše lze vidět trochu jiný přístup k vytváření SWING okna, než u jazyků Java a Scala. V případě použití knihovny SWING u těchto jazyků je obvyklé, že je vytvořena podtřída třídy *JFrame*. V jazyku Clojure není obvyklé vytvářet podtřídy tam, kde se tomu lze vyhnout, proto i v tomto příkladu bylo zvoleno vytvoření okna pomocí vzoru factory. Děje se tak ve funkci *create-window*. Z této funkce je záměrně vypuštěn kód pro přidání panelu a ostatních ovládacích prvků do okna, neboť pro tento příklad nejsou důležité. Stejně tak funkce *create-window-listener* je zjednodušena o implementaci ostatních funkcí rozhraní WindowListener, stejně jako v předchozích ukázkách. Nicméně zde se již nelze vyhnout vytvoření třídy. K tomuto účelu slouží makro *proxy*. Důležitá je v této ukázce především práce se jmennými prostory, kdy makro *„:require“* (řádek 02 ) způsobí načtení jmenného prostoru *p-singleton-clojure.config* a možnost se na něj odvolávat pomocí *„config“*. K prvkům jmenného prostoru pak lze přistoupit tak, že se napíše název a znak „/“, tedy např. *„config/get-default-pah“*. Pokud bychom si místo „/“ představili znak „.“, jednalo by se o prakticky stejný zápis jako přístup ke třídám v jazyku Java a Scala. Dokonce při kompilaci zdrojového kódu je třída pro každý jmenný prostor vytvořena překladačem jazyka Clojure a pokud by výsledný .jar balíček s tímto programem byl přilinkován k programu napsaném v jazyku Java nebo Scala, bylo by možno vidět (a použít) *p-singleton-clojure.config* jako třídu (s názvem *p_singleton_clojure.Config*). Funkce *get-default-path* a *set-default-path* by bylo možno vidět jako funkce s názvem *get_default_path* a *set_default_path*.

### 1.3.2. Builder

Návrhový vzor Builder slouží k oddělení procesu vytváření instance třídy od její reprezentace [9]. Používá se především tehdy, pokud proces vytvoření instance je složitější – např. je potřeba znát pouze některé hodnoty některých z mnoha parametrů dopředu před zahájením procesu a přetížení konstruktoru třídy by nebylo praktické (příliš mnoho parametrů, mnoho různých kombinací povinných a nepovinných).

```
01    public class ComplexClass {
02        private Object part1;
03        private Object part2;
04
05        protected ComplexClass(ComplexClassBuilder builder) {
06            part1 = builder.part1;
07            part2 = builder.part2;
08        }
09
10        public Object getPart1() {
11            return part1;
12        }
13
14        public void setPart1(Object part1) {
15            this.part1 = part1;
16        }
17
18        public Object getPart2() {
19            return part2;
20        }
21
22        public void setPart2(Object part2) {
23            this.part2 = part2;
24        }
25    }
```

*Ukázka kódu 13 Třída vytvářená pomocí builderu v jazyku Java*

V ukázce kódu 13 je vidět příklad třídy, jejíž instance budou vytvářeny pomocí builderu. Za zmínku stojí konstruktor třídy, který je *protected* – není žádoucí, aby konstruktor třídy byl veřejně přístupný, ale je třeba, aby třída *ComplexBuilder*, která je ve stejném balíčku, k němu přístup měla. Tento konstruktor přebírá instanci třídy builder a kopíruje z něj data.

```
01    public class ComplexClassBuilder {
02        protected Object part1;
03        protected Object part2;
04
05        public ComplexClassBuilder(Object part1) {
06            this.part1 = part1;
07        }
08
09        public ComplexClassBuilder setPart2(Object part2) {
10            this.part2 = part2;
11            return this;
12        }
13
14        public ComplexClass build() {
15            return new ComplexClass(this);
16        }
17    }
```

*Ukázka kódu 14 Vzor builder v jazyku Java*

Samotný builder je vidět v ukázce kódu č. 14. V konstruktoru jsou zpravidla předávány parametry, které jsou povinné pro všechny případy procesu vytvoření instance (zde pouze část *part1*). Dále lze vidět metody pro nastavení nepovinných parametrů zpravidla vrací instanci builderu, aby mohlo být využito fluent interface (v ukázce pouze pro část *part2*, která je v některých případech potřebná a v jiných nikoliv).

Poslední metoda, která se na instanci builderu zavolá, se zpravidla jmenuje *build* a vrátí novou instanci vytvářené třídy dle nastavených parametrů.

```
01    Object part1 = new Object();
02    Object part2 = new Object();
03    ComplexClass complexInstance = (new ComplexClassBuilder(part1))
04                                    .setPart2(part2)
05                                    .build();
```

*Ukázka kódu 15 Použití vzoru builder v jazyku Java*

Využití builderu ukazuje ukázka 15. Povinné parametry pro proces vytváření třídy *ComplexClass* jsou předány v konstruktoru builderu, ostatní jsou (či nejsou) předány později pomocí setterů. Nakonec je zavolána metoda *build*.

V jazyku Scala je implementace tohoto návrhového vzoru shodná s jazykem Java, jen se liší syntaxí. Proto není v této práci uveden.

V jazyku Clojure lze využít makra s na první pohled zvláštním názvem „->“. Čte se jako „thread first“[6]. Toto makro přebírá výraz jako první argument a jeho výsledek vloží jako první argument následujících výrazů. Výsledek posledního výrazu je návratová hodnota makra. (Existuje i thread last makro „->>“, které funguje tak, že výsledek prvního výrazu vloží jako poslední argument

[6] Více o thread first makru lze nalézt v dokumentaci jazyka Clojure http://clojuredocs.org/clojure_core/clojure.core/-%3E

následujících výrazů.) Proto zde není třeba vytvářet instanci builderu, ani jej definovat jako třídu – je možno si vystačit se jmenným prostorem.

```
01    (ns p-builder-clojure.complex)
02
03    (defn create-complex
04      [part1]
05      {:part1 part1})
06
07    (defn set-part2
08      [complex part2]
09      (assoc complex :part2 part2))
10
```

*Ukázka kódu 16 Builder v jazyku Clojure*

V ukázce kódu 16 lze vidět implementaci návrhového vzoru Builder v jazyku Clojure. První funkce, *create-complex*, má plní stejnou úlohu, jako konstruktor builderu v Javě a Scale – inicializuje stav builderu a přebírá a ověřuje povinné parametry. V tomto případě vytvoří hashmapu, kterou již lze použít tam, kde jsou očekávána data, která jsou builderem vytvářena. Funkce *set-part2* přebírá jako první argument již hotovou část a rozšiřuje ji.

```
01    (let [part1 (Object.)
02          part2 (Object.)
03          complex (-> (complex/create-complex part1)
04                      (complex/set-part2 part2))])
```

*Ukázka kódu 17 Použití vzoru builder v jazyku Clojure*

V ukázce 17 je vidět, že použití takto definovaného builderu může být stejně snadné a přímočaré, jako v předchozích uvedených jazycích.

### *1.3.2.1 Praktický příklad*

Jako příklad z praxe pro návrhový vzor builder může posloužit např. třída reprezentující data o osobě s názvem *Person*. V případě, že program importuje osobní data z externího zdroje, lze předpokládat, že taková data budou nekompletní. Řekněme, že v našem systému je nutné mít pouze identifikátor osoby, další údaje (v příkladu pro jednoduchost zastoupené pouze jménem a příjmením) nikoliv. V takové situaci lze využít návrhového vzoru builder.

Vzhledem k velké podobnosti tohoto vzoru mezi jazyky Clojure a Scala je v této kapitole příklad v jazyku Scala vynechán. Prakticky by se jednalo o ten samý kód, pouze s jinou syntaxí. Naproti tomu v jazyku Clojure lze dobře využít vlastností tohoto jazyka.

```
01    public class Person {
02        private Integer id;
03        private String name;
04        private String surname;
05
06        protected Person(PersonBuilder builder) {
07            this.id = builder.id
08            this.name = builder.name;
09            this.surname = builder.surname;
10        }
11
12        public Integer getId() {
13            return id;
14        }
15
16        public String getName() {
17            return name;
18        }
19
20        public void setName(String name) {
21            this.name = name;
22        }
23
24        public String getSurname() {
25            return surname;
26        }
27
28        public void setSurname(String surname) {
29            this.surname = surname;
30        }
31
32    }
33    public class PersonBuilder {
34        protected Integer id;
35        protected String name;
36        protected String surname;
37
38        public PersonBuilder(Integer id) {
39            this.id = id;
40        }
41
42        public PersonBuilder setName(String name) {
43            this.name = name;
44            return this;
45        }
46
```

```
47          public PersonBuilder setSurname(String surname) {
48              this.surname = surname;
49              return this;
50          }
51
52          public Person build() {
53              return new Person(this);
54          }
55      }
```
*Ukázka kódu 18 Příklad vzoru builder v jazyku Java*

```
01      Integer id = personInput.getId();
02      PersonBuilder builder = new PersonBuilder(id);
03
04      if (personInput.hasName())
05          builder.setName(personInput.getName());
06
07      if (personInput.hasSurname())
08          builder.setSurname(personInput.getSurname());
09
10      Person person = builder.build();
```
*Ukázka kódu 19 Příklad použití vzoru builder v jazyku Java*

V ukázce kódu 19 lze vidět zjednodušený praktický příklad využití vzoru builder z jazyka Java. Externí vstup zde představuje *personInput*, u kterého máme zaručeno, že vrátí validní identifikátor osoby. Nicméně osobní data již nemusí obsahovat jméno a příjmení. Identifikátor, jakožto povinný parametr, je předán v konstruktoru třídy *PersonBuilder*. Další data, v závislosti na tom, zda-li je vstup obsahuje či nikoliv, jsou předána jako nepovinné parametry přes setter metody builderu. Často se lze setkat i s validací povinných parametrů v konstruktoru builderu, zde je toto pro jednoduchost vynecháno.

```
01      (ns builder-clojure.builder)
02
03      (defn create-person [id]
04        {:id id})
05
06      (defn build-name [partially-built name]
07        (assoc partially-built :name name))
08
09      (defn build-surname [partially-built surname]
10        (assoc partially-built :surname surname))
```
*Ukázka kódu 20 Příklad vzoru builder v jazyku Clojure*

```
01    (defn add-name [part-build input]
02      (if (input/has-name input)
03        (build-name part-build (:name input))
04         part-build))
05
06    (defn add-surname [part-build input]
07      (if (input/has-name input)
08        (build-surname part-build (:surname input))
09         part-build))
10
11
12    …
13    (let [person (-> (create-person (:id input))
14                     (add-name input)
15                     (add-surname input))]
16    … )
```

*Ukázka kódu 21 Příklad použití vzoru builder v jazyku Clojure*

V jazyku clojure je obecně kladen vyšší důraz na využití stávajících jednoduchých datových typů než na vytváření nových tříd. Proto i v příkladu 20 lze vidět, že se data ukládají do hash mapy spíše než do instance třídy. Funkce *create-person* vytvoří hash mapu s povinnými daty. Další funkce, *build-name* a *build-surname* do předané hash mapy doplní nepovinné údaje, je-li to žádoucí. Ukázka 21 znázorňuje využití makra -> pro volání factory metody a nepovinných metod. Aby se zde makro dalo využít a kód zůstal dobře čitelný, je dobré si pomoci funkcemi *add-name* a *add-surname*. Ty jako parametry přebírají částečně vybudovanou hash mapu a vstupní data. V případě, že vstupní data obsahují jméno (příjmení) osoby, přidají je do hash mapy. Pokud ne, jenom vrátí právě budovanou hash mapu.

### 1.3.3. Abstract factory

Dalším zajímavým kreacionálním návrhovým vzorem je abstract factory. Jedná se o vzor, který poskytuje možnost vytvoření instancí třídy (či množiny tříd), přičemž klienta nezajímá konkrétní třída, ale zpravidla pouze implementované rozhraní.

Gamma a spol. uvádí v [9] příklad knihovny pro vytváření uživatelského rozhraní, která podporuje více vzhledů a chování (např. Motif nebo Presentation Manager). Prvky uživatelského rozhraní jsou vytvářeny právě pomocí vzoru abstract factory – programátora vytvářejícího uživatelské rozhraní v tomto případě zajímá, že do formuláře umístí tlačítko. Jak konkrétně bude vypadat, není v době návrhu tolik podstatné. V této kapitole bude využito právě tohoto příkladu (upraveného pro jazyky použité v této práci), nebude proto uveden žádný praktický příklad jako v předchozích kapitolách.

```
01    public enum LookAndFeel {
02        Motif,
03        PM
04    }
05
06    public interface Button {
07        public LookAndFeel getLaF();
08    }
09    public class MotifButton implements Button {
10        @Override
11        public LookAndFeel getLaF() {
12            return LookAndFeel.Motif;
13        }
14    }
15    public class PMButton implements Button {
16        @Override
17        public LookAndFeel getLaF() {
18            return LookAndFeel.PM;
19        }
20    }
21
22    public interface WidgetFactory {
23        public Button createButton();
24    }
25    public class MotifFactory implements WidgetFactory {
26
27        @Override
28        public Button createButton() {
29            return new MotifButton();
30        }
31    }
32    public class PMFactory implements WidgetFactory {
33        @Override
34        public Button createButton() {
35            return new PMButton();
36        }
37    }
```

*Ukázka kódu 22 Implementace návrhového vzoru Abstract Factory v jazyku Java*

V ukázce kódu 22 lze vidět nejprve definici enumerace možných stylů vzhledů a chování. Dále pak definici rozhraní *Button* a dvě třídy implementující toto rozhraní. Pomocí factory bude vytvořena konkrétní třída, avšak klient obdrží pouze objekt implementující právě toto rozhraní, bez specifikace konkrétního typu. Dále následují rozhraní *WidgetFactory* a konkrétní factory pro jednotlivé styly.

```
01    public static WidgetFactory getWidgetFactory(LookAndFeel
configuration) {
02        switch (configuration) {
03            case Motif:
04                return new MotifFactory();
05            case PM:
06                return new PMFactory();
07            default:
08                throw new RuntimeException("factory not found");
09        }
10    }
11
12    public static void main(String[] args) {
13        LookAndFeel configuration = LookAndFeel.Motif;
14        WidgetFactory factory = getWidgetFactory(configuration);
15        Button button = factory.createButton();
16    }
```

*Ukázka kódu 23 Použití abstract factory v jazyku Java*

Ukázka kódu 23 slouží pro rozhodnutí se, kterou konkrétní factory použít na základě konfigurace. Metoda *main* pak ukazuje načtení konfigurace, získání abstract factory a její použití. Je zde vidět, že klient obdrží pouze objekt typu Button, což je rozhraní, nezajímá se o konkrétní třídu. Ta se může lišit na základě konfigurace.

```
01    object LookAndFeel extends Enumeration {
02      type LookAndFeel = Value
03      val Motif, PM = Value
04    }
05
06    trait Button {
07      def getLaF : LookAndFeel.LookAndFeel
08    }
09
10    class MotifButton extends Button {
11      override def getLaF: LookAndFeel.LookAndFeel = LookAndFeel.Motif
12    }
13
14    class PMButton extends Button {
15      override def getLaF: LookAndFeel.LookAndFeel = LookAndFeel.PM
16    }
17
18    trait WidgetFactory {
19      def createButton() : Button
20    }
21
22    class MotifFactory extends WidgetFactory {
23      override def createButton(): Button = new MotifButton
```

```
24    }
25
26    class PMFactory extends WidgetFactory {
27      override def createButton(): Button = new PMButton
28    }
```
*Ukázka kódu 24 Implementace vzoru abstract factory v jazyku Scala*

```
01    object Application extends App {
02      def getWidgetFactory(configuration : LookAndFeel.LookAndFeel) =
03        configuration match {
04          case LookAndFeel.Motif => new MotifFactory
05          case LookAndFeel.PM => new PMFactory
06        }
07
08      val conf = LookAndFeel.Motif
09      val factory = getWidgetFactory(conf)
10      val button = factory.createButton()
11    }
```
*Ukázka kódu 25 Použití vzoru abstract factory v jazyku Scala*

V ukázkách kódu 24 a 25 je vidět, že implementace tohoto návrhového vzoru je v jazyku Scala téměř totožná s jazykem Java. Významný rozdíl je pouze v *getWidgetFactory*, zde je použito pattern matching namísto switch. Díky tomu, že překladač jazyka Scala má informaci o tom, že možné hodnoty parametru *configuration* tvoří konečná množina a zároveň na výsledku této operace závisí návratová hodnota funkce, je překladačem vynuceno, aby zde byly uvedeny buď všechny hodnoty, nebo výchozí případ. Jelikož zde výchozí případ není uveden, další přidání hodnoty do výčtu *LookAndFeel* způsobí chybu při kompilaci. Oproti jazyku Java tak může být kód bezpečnější, aniž by se ztratila pružnost.

```
01    (defrecord Button [laf])
02    (defn create-button-motif []
03      (Button. :motif))
04    (defn create-button-pm []
05      (Button. :pm))
06
07    (defprotocol WidgetFactory
08      (createButton [this]))
09    (defrecord MotifFactory []
10      WidgetFactory
11      (createButton [this] (create-button-motif)))
12    (defrecord PMFactory []
13      WidgetFactory
14      (createButton [this] (create-button-pm)))
15
16    (defmulti get-widget-factory identity)
17    (defmethod get-widget-factory :motif [_]
```

```
18      (MotifFactory.))
19    (defmethod get-widget-factory :pm [_]
20      (PMFactory.))
21
22    (defn main []
23      (let [factory (get-widget-factory :motif)
24            button  (.createButton factory)]))
```

*Ukázka kódu 26 Návrhový vzor Abstract Factory v jazyku Clojure*

V ukázce kódu 26 lze vidět variantu Abstract factory v jazyku Clojure, která využívá definice typů. Výsledná hodnota kódu (uložená v *button*) tak bude typu `abstract_factory_clojure.core.Button` namísto `clojure.lang.PersistentArrayMap`, což je typ, který se v jazyku Clojure nejčastěji používá na uchovávání dat. Tato vlastnost se může hodit především tehdy, je-li potřeba pracovat s kódem napsaným v jiných, ne dynamicky typovaných jazycích. Podobně u factory je také vždy známý typ, na který se lze reflexí dotázat.

Namísto techniky pattern matching, která byla použita v Scala kódu, zde byly použity multimetody. Jedná se o způsob, jakým je v jazyku Clojure implementováno přetěžování metod, nicméně tento koncept je o něco obecnější. Přetěžování metod zpravidla funguje pouze na základě arity metod (v dynamicky typovaných jazycích) nebo na základě arity a typu vstupních parametrů (ve staticky typovaných jazycích). V jazyku Clojure je rozhodnutí prováděno pomocí programátorem definované funkce (tzv. dispatch funkce). Ta je předána jako vstupní parametr makra *defmulti* a jí jsou předány všechny vstupní parametry multimetody při jejím zavolání. Na základě výsledku dispatch funkce jsou potom volány konkrétní funkce. Její výsledek je porovnán s druhým parametrem makra *defmethod*, vždy nejprve pomocí funkce = (je shodný) a pokud není nalezen, je použita funkce *isa?*, která zkoumá nejen shodnost, ale i hierarchii[7]. V tomto případě byla jako dispatch funkce využita *identity*, což je funkce, která vrací vstupní parametry.
V ukázaném případě se jedná o dostatečné řešení. Jediným vstupním parametrem multimetody *get-widget-factory* je klíčové slovo označující vzhled a chování požadovaného tlačítka.

Pokud by nebylo vyžadováno, aby byly typy factory a produktu staticky definovány, lze kód zjednodušit.

```
01    (defn create-button-motif []
02      {:laf :motif})
03    (defn create-button-pm []
04      {:laf :pm})
05
06    (defmulti get-widget-factory identity)
07    (defmethod get-widget-factory :motif [_]
08      create-button-motif)
09    (defmethod get-widget-factory :pm [_]
10      create-button-pm)
11
12    (defn main []
```

[7] Více informací na toto téma lze nalézt v dokumentaci http://clojure.org/multimethods

```
13      (let [factory (get-widget-factory :motif)
14            button  (factory)]))
```
*Ukázka kódu 27 Zjednodušená implementace vzoru Abstract factory v jazyku Clojure*

Zde již bude typ hodnoty v *button* `clojure.lang.PersistentArrayMap`. To je nepříjemné, pokud je potřeba vytvořený produkt předat kódu, který je staticky typovaný. Nicméně pokud se s produktem bude pracovat pouze z jazyka Clojure a nebude vadit „ztráta“ informace o tom, že se jedná o tlačítko, je možno použít tuto variantu. Další změnou je chování multimetody *get-widget-factory*. Místo toho, aby vracela objekt, vrátí přímo funkci na vytvoření tlačítka.

## 1.4. Behaviorální vzory

Zatímco strukturální vzory popsané v [9] jsou velmi spjaté s objektově orientovaným programováním a nemá smysl se jimi v této práci zabývat, některé behaviorální jsou důležité i ve funkcionálním programování – úlohy, které tyto vzory řeší, je třeba ve funkcionálním programování řešit také. Z návrhových vzorů v [9] byly vybrány vzory Command, Iterator, Observer, Strategy a Template method, jejichž využití má smysl i ve funkcionálním programování. Ostatní vzory jsou buď natolik spjaty s objektově orientovaným programováním, že při jejich implementaci není možno využít funkcionálních prvků, nebo řeší situace, které by ve funkcionálním programování neměly nastat (např. vzor memento, který zachycuje vnitřní stav objektu tak, aby daný objekt mohl být do takového stavu opět uveden – funkcionální programování klade důraz především na bezstavovost).

### 1.4.1. Command

Jedná se o návrhový vzor, který má za cíl zaobalit jistou funkcionalitu jako objekt tak, aby mohla být vykonána později. Časté použití je např. při vytváření uživatelského rozhraní, kdy reakce na události vyvolané uživatelem jsou implementovány právě pomocí tohoto vzoru.

```
01    public interface ICommand {
02        public void execute();
03    }
04    public class Command implements ICommand {
05        @Override
06        public void execute() {
07            System.out.println("Hello World!");
08        }
09    }
```

*Ukázka kódu 28 Command v jazyku Java*

```
01    invoker.command = new Command();
```

*Ukázka kódu 29 Použití Command v jazyku Java*

V jazyku Java je vzor command implementován zpravidla tak, jak je vidět z ukázky kódu 28. Je definováno rozhraní (*ICommand* v ukázce) a volající kód (v [9] označen jako *Invoker*) očekává předání objektu implementujícího toto rozhraní. Volající kód následně při vhodné příležitosti zavolá metodu *execute* objektu *Command*.

V jazycích Scala a Clojure je možné tento vzor rovněž implementovat touto cestou. Ale díky tomu, že se zde mohou vyskytovat funkce samostatně, aniž by musely být součástí objektu, je možno využít této vlastnosti a namísto instance celého objektu předat do volajícího kódu pouze funkci či lambda výraz[8]. Není tak třeba definovat nové rozhraní a implementovat jej, což vede ke zkrácení a zpřehlednění kódu. Toto řešení je blíže typickému řešení z procedurálního kódu, např. C, kde se běžně předává ukazatel na funkci, často označováno jako callback funkce. Jelikož se jedná o základní vlastnost funkcionálních jazyků, její použití je zřejmé a proto zde není uveden příklad.

[8] Toto je od verze 8 možné provést i v jazyku Java.

### 1.4.2. Iterator

Tento návrhový vzor má za cíl poskytnout sekvenční přístup k prvkům kolekce, bez zveřejnění vnitřní reprezentace kolekce. [9]

Iterator je poměrně často používán při procházení kolekcí nebo stromových struktur. V ukázkové implementaci (ukázka kódu 30) představuje třída *List* právě kolekci. Tato kolekce podporuje některé základní operace, např. zjištění velikosti a přístup k náhodnému prvku. Rozhraní *Iterator* definuje operace, které může provést iterátor – první zjistí, zdali je možno se posunout na další prvek, druhá posune „kurzor“ na další prvek a vrátí jej. Třída *ListIterator* představuje konkrétní iterátor nad kolekcí v příkladu. Instance iterátoru pro klienta je vytvářena pomocí metody *iterator* v kolekci.

```
02    public class List<TElement> {
03        public int getSize() { return 0; }
04        public TElement getElement(int idx) { return null; }
05
06        public ListIterator<TElement> iterator() {
07            return new ListIterator<TElement>(this);
08        }
09    }
10    public interface Iterator<TElement> {
11        public boolean hasNext();
12        public TElement next();
13    }
14
15    public class ListIterator<TElement> implements Iterator<TElement>
{
16        private List<TElement> list;
17        private int index = 0;
18
19        public ListIterator(List<TElement> list) {
20            this.list = list;
21        }
22
23        @Override
24        public boolean hasNext() { return index < list.getSize(); }
25
26        @Override
27        public TElement next() {
28            if (hasNext()){
29                index++;
30                return list.getElement(index);
31            }
32        }
33    }
```

*Ukázka kódu 30 Příklad implementace vzoru Iterator v jazyku Java*

V [9] je popsáno několik variant tohoto návrhového vzoru. Za zmínku v této práci stojí především *externí* a *interní* iterátory. V příkladu dostane klient přímo instanci iterátoru, se kterou může manipulovat, může se přímo dotazovat na stav iterátoru a získává referenci na jeden prvek po druhém. Toto je **externí** iterátor. Jedná se o nejčastější variantu tohoto vzoru v objektově orientovaných jazycích.

Naproti tomu **interní** iterátor nenechává odpovědnost za kontrolu stavu a projití kolekce prvek po prvku na klientovi, nýbrž zpravidla přebírá funkci (či funkční objekt implementující předepsané rozhraní) a o správné projití kolekce se stará sám. Toto je nejčastější varianta iterátoru ve funkcionálních jazycích.

```
01    class ForEachIterator[T](list : List[T]) {
02
03      def iterate(func : (T) => Unit) = {
04
05        def recurIterate(aList : List[T]) : Unit = {
06          if (aList.isEmpty)
07            return
08
09          func(aList.head)
10          recurIterate(aList.tail)
11        }
12
13        recurIterate(list)
14      }
15    }
16
17    object Main extends App{
18      val list = List('H', 'e', 'l', 'l', 'o')
19      val iterator = new ForEachIterator(list)
20
21      iterator.iterate(c => print(c))
22    }
```

*Ukázka kódu 31 Iterátor v jazyku Scala*

V ukázce kódu 31 je ukázána implementace interního iterátoru, který jednoduše projde všechny prvky kolekce za sebou (zde vyjádřeno rekurzí pomocí vnitřní funkce *recurIterate*; Scala podporuje tzv. tail-call optimalizaci při rekurzi – rekurze se používá mnohem častěji než v jazycích, které toto nepodporují). Mezi hlavní výhody interních iterátorů patří kratší a srozumitelnější kód, mezi nevýhody potom to, že samotný algoritmus určující pořadí projití prvků je implementován přímo v třídě pro iterátor, ne až v klientovi, což snižuje flexibilitu kódu. Naštěstí v základních knihovnách jazyků Scala a Clojure je množství iterátorů připravených a efektivně pokrývajících mnoho případů užití. Ve většině případů tedy stačí napsat pouze klientský kód, který je zpravidla kratší než v případě použití externích iterátorů.

```
01    (defn iterate-through [coll f]
02      (loop [c coll]
03        (if-not (empty? c)
```

```
04          (do (f (first c))
05              (recur (rest c))))))
06
07    (defn main []
08      (let [coll '("h", "e", "l", "l", "o")]
09        (iterate-through coll #(println %1))))
```
*Ukázka kódu 32 Implementace vzoru Iterator v jazyku Clojure*

Stejného způsobu využívá i kód v ukázce 32. Rozdíl je v tom, že jazyk Clojure nepodporuje tail-call optimalizaci rekurze, proto je v iterátoru (zde implementován jako funkce, nikoliv třída) použita forma loop-recur. Díky této formě je možno vyjádřit rekurzi, aniž by se riskovalo přetečení zásobníku. Forma *#( ... )* je v jazyku clojure zkrácený zápis pro anonymní funkci, sémanticky ekvivalentní
s *(fn [...] ...)*.

Prakticky se tento vzor v jazycích Clojure a Scala používá všude, kde je potřeba pracovat s kolekcemi a jeho použití je jednodušší než v jazyku Java (většinou se jedná o interní iterátory a klientský kód se tedy stává z 1 řádku). Proto není třeba uvádět praktické příklady.

### 1.4.3. Observer

Observer cílí na situaci, kdy objekt (*Observable*) potřebuje poslat zprávu jiným objektům (*Observers*) o změně svého stavu. S tímto vzorem je možné se setkat nejčastěji při implementaci uživatelských rozhraní, kde jeden z prvků obdrží vstup od uživatele a dá zprávu ostatním prvkům (např. vyplnění textového vstupního pole). [9]

```
01    public interface Observable {
02        void addObserver(Observer observer);
03        void removeObserver(Observer observer);
04        void notifyObservers();
05    }
06    public interface Observer {
07        public void onObservableStateChanged(Observable observable);
08    }
09    public class Subject implements Observable {
10        final List<Observer> changesObservers = new ArrayList<>(1);
11
12        @Override
13        public void addObserver(Observer observer) {
14            changesObservers.add(observer);
15        }
16
17        @Override
18        public void removeObserver(Observer observer) {
19            changesObservers.remove(observer);
20        }
21
```

```
22        @Override
23        public void notifyObservers() {
24            for (Observer observer : changesObservers) {
25                observer.onObservableStateChanged(this);
26            }
27        }
28    }
29    public class Client implements Observer {
30        private Observable observable;
31
32        public Client() {
33            observable = new Subject();
34            observable.addObserver(this);
35        }
36
37        @Override
38        public void onObservableStateChanged(Observable observable) {
39            System.out.println("state changed");
40        }
41    }
```

*Ukázka kódu 33 Observable v jazyku Java 7 a nižších verzích*

V ukázce 33 je vidět tradiční implementace tohoto vzoru v jazyku Java. Základem jsou 2 rozhraní, *Observable* pro objekt, který notifikuje ostatní o změně stavu a *Observer*, který je notifikován. Objektem, jehož stav se změní, je instance třídy *Subject* a *Client* dostane notifikaci o změně. Je zde vidět, že *Client* se musel k odběru „přihlásit“ (řádek 34 ). Nevýhodou této implementace je skutečnost, že je velmi náročná na množství kódu, který se musí přidat do obou zainteresovaných objektů. Ten se sice dá napsat v bázové třídě, ale ne vždy je to možné či žádoucí, proto je časté jeho kopírování.

Jazyk Java verze 8 přidala podporu lambda výrazů a výchozích implementací metod v rozhraních. Toho se dá využít pro zjednodušení implementace:

```
01    public interface Observable8 {
02        final List<Consumer<Observable8>> changesObservers = new
ArrayList<>(1);
03
04        default void addObserver(Consumer<Observable8> func) {
05            changesObservers.add(func);
06        }
07
08        default boolean removeObserver(Consumer<Observable8> func){
09            return changesObservers.remove(func);
10        }
11
12        default void notifyObservers(){
```

```
13            List<Consumer<Observable8>> observers =
changesObservers.stream()
14                    .filter((c) -> (c != null))
15                    .collect(Collectors.toList());
16            observers.forEach(o -> o.accept(this));
17        }
18    }
19    public class Subject8 implements Observable8 {
20        public void changeState() {
21            notifyObservers();
22        }
23    }
24    public class Client8 {
25        private Observable8 observable;
26
27        public Client8() {
28            observable = new Subject8();
29            observable.addObserver(observable8 ->
System.out.println("state changed"));
30        }
31    }
```

*Ukázka kódu 34 Implementace vzoru Observer v jazyce Java verze 8*

Zde se mírně zesložitila implementace rozhraní *Observable*, které obsahuje samotný kód. Ten již nemusí být přítomen ve třídě *Subject* – její úloha se (v tomto vzoru) změnila pouze na zavolání příslušné metody implementované v rozhraní tehdy, kdy je potřeba. Dále bylo možno využít rozhraní *Consumer*, které je součástí základní knihovny jazyka Java (balíčku java.util.function) a de facto se jedná o rozhraní pro funkci přebírající 1 parametr s návratovým typem void (tedy „zkonzumuje parametr a nic nevyprodukuje“). Instanci objektu implementujícího toto rozhraní je možné vytvořit pomocí lambda výrazu. Dobrou zprávou ale je, že rozhraní *Observable* je znovupoužitelné.

```
01    trait Observable {
02      var observers : List[Observable => Unit] = Nil
03
04      def addObserver(observer : Observable => Unit) =
05         observers ::= observer
06
07      def removeObserver(observer : Observable => Unit) =
08         observers = observers.filterNot(o => o == observer)
09
10      def notifyObservers() =
11         observers.foreach(observer => observer.apply(this))
12    }
13
14    class Subject extends Observable {
15      def changeState() = notifyObservers()
```

```
16    }
17
18    class Client {
19      val observable = new Subject()
20
21      observable.addObserver(observable => println(observable))
22    }
```
*Ukázka kódu 35 Implementace vzoru observer v jazyku Scala*

V ukázce 35 lze vidět implementaci vzoru observer v jazyku Scala, která se nijak zvlášť neliší od implementace v jazyku Java verze 8. Rozdíl zde je v tom, že *Observable* není definováno jako rozhraní, ale *trait*. Lze jej vidět jako náhražku rozhraní z jazyka Java (verze 8, tedy s možností implementovat metody). Rozhraní *Observer* v tomto příkladu, stejně jako v předchozím, chybí, neboť funkce *addObserver* (a *removeObserver)* z *Observable* přebírají pouze funkci. Za zmínku také stojí volání *observers.filterNot* (v metodě *removeObserver*) a *observers.foreach* (v metodě *notifyObservers*). Jedná se o použití interních iterátorů, viz předchozí kapitola o návrhovém vzoru iterator.

```
01    (def observable (ref nil))
02
03    (add-watch observable :observable-watcher
04               (fn [key ref old new] (do (println "Was" old)
05                                         (println "Now is" new))))
06
07    (dosync (ref-set observable "Hello World"))
```
*Ukázka kódu 36 Implementace vzoru observer v jazyku Clojure*

Naopak kód jazyka Clojure (ukázka kódu 36) je zcela jiný. Jelikož Clojure je primárně funkcionální jazyk, zpravidla v programech napsaných v Clojure nejsou třídy a objekty využívány v takové míře. Data jsou zpravidla uloženy v hash mapách či seznamech (na což naráží i zkratka LISP = List Processing). Jelikož v Clojure je kladen velký důraz na neměnnost stavu a o data, jejichž stav lze změnit, se primárně stará software transactional memory, je možné tohoto využít – STM v Clojure nabízí funkce *add-watch* a *remove-watch*[9], kde je možno definovat jaká funkce se má zavolat při změně stavu. Samotná volaná funkce má 4 argumenty – klíč (identifikátor) klienta, referenci, jejíž stav se změnil, starou hodnotu a novou hodnotu. Toto poskytuje pohodlnou náhradu vzoru observer, která pokrývá většinu případů užití. Bohužel funkce *add-watch* a *remove-watch* jsou v době psaní této práce v alfa verzi a jejich chování či rozhraní se může změnit.

### *1.4.3.1 Praktický příklad*

Dobrý příklad vzoru observer je možno najít v uživatelských rozhraních. U některých desktop aplikací je možno v jednom pohledu mít zobrazená stejná data, která jsou právě editována v jiném pohledu. V tomto případě jsou data reprezentována třídou *Person*, data jsou zobrazena v *PersonViewPanel* a editována v *PersonEditPanel*.

[9] Více informací o těchto funkcích je možno nalézt v dokumentaci jazyka Clojure:
http://clojuredocs.org/clojure_core/clojure.core/add-watch
http://clojuredocs.org/clojure_core/clojure.core/remove-watch

```
01    public interface Observable {
02        void addObserver(Observer observer);
03        void removeObserver(Observer observer);
04        void notifyObservers();
05    }
06    public interface Observer {
07        public void onObservableStateChanged(Observable observable);
08    }
09    public class Person implements Observable {
10        private String name;
11        private String surname;
12
13        public String getName() {
14            return name;
15        }
16
17        public void setName(String name) {
18            this.name = name;
19            notifyObservers();
20        }
21
22        public String getSurname() {
23            return surname;
24        }
25
26        public void setSurname(String surname) {
27            this.surname = surname;
28            notifyObservers();
29        }
30
31        private List<Observer> observers = new ArrayList<Observer>();
32
33        @Override
34        public void addObserver(Observer observer) {
35            observers.add(observer);
36        }
37
38        @Override
39        public void removeObserver(Observer observer) {
40            observers.remove(observer);
41        }
42
43        @Override
44        public void notifyObservers() {
45            for (Observer observer : observers) {
46                observer.onObservableStateChanged(this);
```

```
47            }
48        }
49    }
50
51    public class PersonViewPanel extends JPanel {
52        JLabel personName;
53        JLabel personSurname;
54
55        Person person;
56        Observer personObserver;
57
58        private void setPerson(Person person) {
59            if (this.person != null && this.personObserver != null)
60                person.removeObserver(personObserver);
61
62            this.person = person;
63            if (person == null) {
64                personObserver = null;
65            } else {
66                this.personObserver = new Observer() {
67                    @Override
68                    public void onObservableStateChanged(Observable
observable) {
69                        updatePerson();
70                    }
71                };
72            }
73
74            updatePerson();
75        }
76
77        private void updatePerson() {
78            if (person == null)  {
79                personName.setText("");
80                personSurname.setText("");
81            } else {
82                personName.setText(person.getName());
83                personSurname.setText(person.getSurname());
84            }
85        }
86
87        // ...
88
89
90    }
91    public class PersonEditPanel extends JPanel {
```

```
92        private JTextField personName;
93        private JTextField personSurname;
94
95        private Person person;
96
97        public PersonEditPanel() {
98            // ...
99            personName.getDocument().addDocumentListener(new
DocumentListener() {
100               @Override
101               public void insertUpdate(DocumentEvent e) {
102                   updatePersonName();
103               }
104
105               @Override
106               public void removeUpdate(DocumentEvent e) {
107                   updatePersonName();
108               }
109
110               @Override
111               public void changedUpdate(DocumentEvent e) {
112                   updatePersonName();
113               }
114
115               private void updatePersonName() {
116                   person.setName(personName.getText());
117               }
118           });
119
120           personSurname.getDocument().addDocumentListener(new
DocumentListener() {
121               @Override
122               public void insertUpdate(DocumentEvent e) {
123                   updatePersonSurName();
124               }
125
126               @Override
127               public void removeUpdate(DocumentEvent e) {
128                   updatePersonSurName();
129               }
130
131               @Override
132               public void changedUpdate(DocumentEvent e) {
133                   updatePersonSurName();
134               }
135
```

```
136                 private void updatePersonSurName() {
137                     person.setName(personName.getText());
138                 }
139             });
140 
141             //...
142         }
143     }
```

*Ukázka kódu 37 Příklad vzoru Observer v jazyku Java*

V ukázce kódu 37 lze vidět příklad takové situace v jazyku Java. Kód byl zkrácen na ty části, které se týkají vzoru observer. Části jako inicializace grafických prvků byly vynechány. Třída *Person* implementuje rozhraní *Observable* a při zavolání *setName* a *setSurname* notifikuje všechny zaregistrované objekty o změně stavu. *PersonViewPanel* je komponenta, která zobrazuje data o osobě v polích *personName* a *personSurname*. Pomocí metody *setPerson* je nastavena osoba, jejíž údaje tento panel zobrazuje. Zároveň je v této metodě *PersonViewPanel* zaregistrován jako observer pro třídu *Person*. Samotná implementace rozhraní *Observer* je zde řešena jako vnitřní anonymní třída a reference na její instanci je uložena, aby bylo později možno ji odregistrovat ze seznamu observerů. V případě, že nelze bezpečně zajistit odregistrování, lze v třídě implementující *Observable* využít třídu *java.lang.ref.WeakReference*[10], čímž lze předejít situaci, kdy zaregistrovaný observer již není potřeba a je zbytečně držen garbage collectorem v paměti (protože na něj existuje reference z třídy implementující *Observable*). Zmíněná implementace rozhraní *Observer* reaguje na změnu stavu sledovaného objektu tím, že zavolá metodu *updatePerson*, která aktualizuje údaje v textových polích. Třída *PersonEditPanel* představuje grafickou komponentu, která umožňuje editovat údaje o osobě pomocí vstupních polí *personName* a *personSurname*. V případě, že uživatel upraví obsah těchto polí, je zavolána příslušná *set* metoda objektu *Person*. Ta notifikuje všechny objekty v roli observer, včetně instance *PersonViewPanel*, která aktualizuje zobrazovaná data. Tím je zajištěno, že uživatel má ve všech otevřených pohledech vždy aktuální data.

```
01    trait Observable {
02      var observers : List[Observable => Unit] = Nil
03 
04      def addObserver(observer : Observable => Unit) = observers ::=
observer
05      def removeObserver(observer : Observable => Unit) = observers =
observers.filterNot(o => o == observer)
06 
07      def notifyObservers() = observers.foreach(observer =>
observer.apply(this))
08    }
09 
10    class Person extends Observable {
11      private var name : String = ""
```

[10] Více informací o třídě *WeakReference* z balíčku *java.lang.ref* lze najít v dokumentaci jazyka Java. http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html

```
12      private var surname : String = ""
13
14      def getName : String = name
15      def setName(name : String) : Unit = {
16        this.name = name
17        notifyObservers()
18      }
19
20      def getSurname : String = surname
21      def setSurname(surname : String) : Unit = {
22        this.surname = surname
23        notifyObservers()
24      }
25    }
26
27    class PersonViewPanel {
28      private var personName : JTextField = new JTextField()
29      private var personSurname : JTextField = new JTextField()
30
31      private var person : Person = null
32      private var personObserver : (Observable => Unit) = null
33
34      private def setPerson(person: Person) {
35        if (this.person != null && this.personObserver != null)
36          person.removeObserver(personObserver)
37
38        this.person = person
39        if (person == null) {
40          personObserver = null
41        }
42        else
43        {
44          this.personObserver = o => updatePerson
45        }
46        updatePerson
47      }
48
49      private def updatePerson {
50        if (person == null) {
51          personName.setText("")
52          personSurname.setText("")
53        }
54        else {
55          personName.setText(person.getName)
56          personSurname.setText(person.getSurname)
57        }
```

```
58      }
59
60    }
61
62    class PersonEditPanel extends JPanel {
63      private var personName : JTextField = new JTextField()
64      private var personSurname : JTextField = new JTextField()
65
66      private var person : Person = null
67
68      // other initializations
69
70      personName.getDocument.addDocumentListener(new DocumentListener
{
71        override def insertUpdate(e: DocumentEvent) = updatePerson()
72        override def changedUpdate(e: DocumentEvent) = updatePerson()
73        override def removeUpdate(e: DocumentEvent) = updatePerson()
74
75        private def updatePerson() =
person.setName(personName.getText)
76      })
77
78      personSurname.getDocument.addDocumentListener(new
DocumentListener {
79        override def insertUpdate(e: DocumentEvent) = updatePerson()
80        override def changedUpdate(e: DocumentEvent) = updatePerson()
81        override def removeUpdate(e: DocumentEvent) = updatePerson()
82
83        private def updatePerson() =
person.setSurname(personName.getText)
84      })
85    }
```

*Ukázka kódu 38 Příklad vzoru Observer v jazyku Scala*

Ukázka 38 představuje výše popsaný příklad v jazyku Scala. *Observable* je zde implementováno jako *trait*, tedy může obsahovat i proměnné a metody. Třída *Person*, na rozdíl od příkladu v jazyky Java, neobsahuje seznam observerů a implementaci metod tohoto rozhraní – ty jsou v tomto případě obsaženy již v *Observable*. Toto by bylo možné i v jazyku Java verze 8, kde rozhraní může obsahovat výchozí implementace metod. Třídy *PersonViewPanel* a *PersonEditPanel* jsou velmi podobné předchozímu příkladu. Největším a nejpodstatnějším rozdílem je absence rozhraní *Observer* (a jeho implementace), které je řešeno prostou funkcí.

Vzhledem k faktu, že jazyk Clojure dává důraz na neměnnost dat, tedy přesný opak toho, jaké situace řeší návrhový vzor observer, praktický příklad v jazyku Clojure zde není uveden. Nicméně bylo by možno využít objektově orientovaných vlastností tohoto jazyka a vytvořit kód velmi podobný kódu v jazyku Java (místo rozhraní by se vytvořil protokol pomocí funkce *defprotocol*, dále pro implementaci metod protokolu lze využít maker *extend-type* a *reify*).

### 1.4.4. Strategy

Pomocí vzoru strategy lze dynamicky, za běhu programu, změnit část funkcionality programu. Jedná se o často vyskytující se vzor, bohužel je vždy potřeba napsat množství boilerplate kódu, což ne všichni programátoři oceňují. Jelikož se jedná o poměrně jednoduchý vzor, není třeba jej uvádět poprvé v čisté a podruhé v „praktické“ formě.

```
01    public interface Algorithm {
02        int perform(int a, int b);
03    }
04
05    public class Add implements Algorithm {
06        public int perform(int a, int b) {
07            return a + b;
08        }
09    }
10
11    public class Substract implements Algorithm {
12        public int perform(int a, int b) {
13            return a - b;
14        }
15    }
16
17    public class Main {
18        public static int getResult(Algorithm strategy, int a, int b)
{
19            return strategy.perform(a, b);
20        }
21
22        public static void main(String[] args) {
23            int sum = getResult(new Add(), 10, 5);
24            int difference = getResult(new Substract(), 10, 5);
25        }
26    }
```

*Ukázka kódu 39 Implementace vzoru strategy v jazyku Java*

V ukázce 39 je definováno rozhraní *Algorithm*, které zde vyjadřuje algoritmus pro práci s čísly. Třídy *Add* a *Substract* vyjadřují operace sčítání a odčítání, implementují rozhraní *Algorithm*. V metodě *getResult* vystupuje konkrétní algoritmus právě jako strategy.

V jazyku Scala, Clojure a v jazyku Java od verze 8 již není třeba definovat rozhraní a třídy pro jednotlivé strategy – pokud daná strategy s sebou nenese žádný stav, je možno ji nahradit přímo funkcemi.

```
01    class Main extends App {
02
03      val addOperation = (a : Int, b : Int) => a + b
04      val subOperation = (a : Int, b : Int) => a - b
```

```
05
06      def getResult(strategy : (Int, Int) => Int, a : Int, b : Int) =
07        strategy(a, b)
08
09      val sum = getResult(addOperation, 10, 5)
10      val difference = getResult(subOperation, 10, 5)
11    }
```
*Ukázka kódu 40 Implementace vzoru strategy v jazyku Scala*

```
01    (def addOperation #(+ %1 %2))
02    (def subOperation #(- %1 %2))
03
04    (defn getProduct [strategy a b]
05      (strategy a b))
06
07    (def sum (getProduct addOperation 10 5))
08    (def difference (getProduct subOperation 10 5))
```
*Ukázka kódu 41 Implementace vzoru strategy v jazyku Clojure*

Samozřejmě, pokud je potřeba, aby strategy pracovala se svým vnitřním stavem mezi jednotlivými použitími, je potřeba je i v jazycích Java 8, Scala a Clojure definovat jako objekty. V mnoha případech to však není nutné (funkce si může nesdílený stav udržovat ve svých lokálních proměnných a volat jiné funkce, což je často dostačující).

### 1.4.5. Template method

Tento vzor definuje kostru algoritmu, přičemž jeho některé kroky mohou být předefinovány v podtřídách, nicméně samotná kostra algoritmu předefinována být nemůže. [9] Jedná se o vzor, jehož implementace je velmi podobná vzoru Strategy. Pro velkou podobnost s tímto vzorem zde také není potřeba uvádět praktický příklad

```
01    public class Algorithm {
02        public void doStep1() {
03            System.out.println("default step 1");
04        }
05
06        public void doStep2() {
07            System.out.println("default step 2");
08        }
09
10        public void doStep3() {
11            System.out.println("default step 3");
12        }
13    }
14    public class CustomizedAlgorithm extends Algorithm{
15        @Override
16        public void doStep2() {
```

```
17              System.out.println("customized step 2");
18          }
19      }
20      public class Client {
21          private Algorithm algorithm;
22
23          public Client(Algorithm algorithm) {
24              this.algorithm = algorithm;
25          }
26
27          public void performAlgorithm() {
28              algorithm.doStep1();
29              algorithm.doStep2();
30              algorithm.doStep3();
31          }
32      }
33      public class Main {
34
35          public static void main(String[] args) {
36              CustomizedAlgorithm customizedAlgorithm =
37                  new CustomizedAlgorithm();
38              Client client = new Client(customizedAlgorithm);
39              client.performAlgorithm();
40          }
41      }
```

*Ukázka kódu 42 Implementace vzoru Template Method v jazyku Java*

Zde je ve třídě *Algorithm* definována kostra algoritmu (jeho jednotlivé kroky) a výchozí implementace těchto kroků. Třída *CustomizedAlgorithm* předefinuje potřebné kroky. *Client* provede kroky algoritmu ve správném pořadí bez ohledu na to, jestli byly předefinovány, či nikoliv.

Implementace tohoto vzoru by v jazyku Scala byla téměř stejná, proto je vynechána.

```
01      (def def-step1 (fn [] (println "default step 1")))
02      (def def-step2 (fn [] (println "default step 2")))
03      (def def-step3 (fn [] (println "default step 3")))
04
05      (defn create-algorithm
06        [{:keys [step1 step2 step3]
07            :or {step1 def-step1 step2 def-step2 step3 def-step3}}]
08        {:step1 step1 :step2 step2 :step3 step3})
09
10      (defn create-custom-algorithm []
11        (let [custom-step-2 (fn [] (println "custom step 2"))]
12          (create-algorithm {:step2 custom-step-2})))
13
```

```
14    (defn client [algorithm]
15      (do ((:step1 algorithm))
16          ((:step2 algorithm))
17          ((:step3 algorithm))))
18
19    (defn main []
20      (let [algorithm (create-custom-algorithm)]
21        (client algorithm)))
```

*Ukázka kódu 43 Implementace vzoru Template Method v jazyku Clojure*

V ukázce kódu 43 je nejprve vidět výchozí implementace kroků (*def-step1 – 3*). Továrnička *create-algorithm* poté přebírá jako argument mapu kde klíče jsou *:step1, :step2, :step3* a hodnoty jednotlivé funkce pro vykonání daných kroků. Pokud některý z těchto klíčů není předán, použije se na jeho místě výchozí implementace. Továrnička *create-custom-algorithm* tedy předefinuje pouze krok 2 algoritmu. Klient je v tomto případě opět funkce, která přebírá algoritmus nyní jako mapu a ve správném pořadí zavolá funkce, které jsou v mapě uloženy jako hodnoty. V mapě reprezentující algoritmus nemusí být uloženy pouze funkce pro jednotlivé kroky algoritmu, ale také data pro jejich provedení, jsou-li potřeba.

## 1.5. Funkcionální vzory

Zatímco na poli objektově orientovaného programování existuje dobře známá publikace [9], která v sobě shrnuje důležité návrhové vzory, na poli funkcionálního programování se mi nepodařilo takovou publikaci najít. Nicméně i zde lze v programech objevit často jednotný přístup k řešení různých úloh.

### 1.5.1. Zpracování kolekcí: Filter-Map-Reduce

V business aplikacích se lze často setkat se zpracováváním kolekcí dat. Například i v triviální aplikaci jako je internetový obchod je třeba při výpočtu cen „vytáhnout“ z košíku zboží, které spadá do dané kategorie DPH, spočítat DPH a spočítat konkrétní cenu. Tento postup bude demonstrován přímo na praktickém příkladu.

```
01    import java.util.*;
02
03    public class Main {
04
05        private static class BasketItem {
06            private int price;
07
08            private BasketItem(int id) {
09                this.price = id;
10            }
11
12            public int getPrice() {
13                return price;
14            }
15
16            public void setPrice(int price) {
17                this.price = price;
18            }
19
20            @Override
21            public String toString() {
22                return "BasketItem{" +
23                        "price=" + price +
24                        '}';
25            }
26
27            @Override
28            public boolean equals(Object o) {
29                if (this == o) return true;
30                if (o == null || getClass() != o.getClass()) return
false;
31
```

```
32                BasketItem that = (BasketItem) o;
33
34                if (price != that.price) return false;
35
36                return true;
37            }
38
39            @Override
40            public int hashCode() {
41                return price;
42            }
43        }
44
45        private static List<BasketItem> createBasket() {
46            ArrayList<BasketItem> list = new ArrayList<BasketItem>();
47            list.add(new BasketItem(100));
48            list.add(new BasketItem(200));
49            list.add(new BasketItem(300));
50            list.add(new BasketItem(400));
51            list.add(new BasketItem(500));
52
53            return Collections.unmodifiableList(list);
54        }
55
56        private static List<BasketItem>
getHighVatItems(List<BasketItem> basketItems) {
57            ArrayList<BasketItem> highVat = new
ArrayList<BasketItem>();
58            for (BasketItem it : basketItems) {
59                if (it.getPrice() >= 300)
60                    highVat.add(it);
61            }
62
63            return Collections.unmodifiableList(highVat);
64        }
65
66        private static Map<BasketItem, Long>
calculateHighVat(List<BasketItem> items) {
67            Map<BasketItem, Long> vatmap = new HashMap<BasketItem,
Long>();
68            for (BasketItem it : items) {
69                long pricePlusVat = Math.round(it.getPrice() +
(it.getPrice() * 0.2));
70                vatmap.put(it, pricePlusVat);
71            }
72
```

```
73              return Collections.unmodifiableMap(vatmap);
74          }
75
76          private static long sumPrices(Map<BasketItem, Long> items) {
77              long sum = 0;
78              for (Map.Entry<BasketItem, Long> item : items.entrySet())
{
79                  sum += item.getValue();
80              }
81
82              return sum;
83          }
84
85          public static void main(String[] args) {
86              List<BasketItem> allItems = createBasket();
87              List<BasketItem> highVatItems = getHighVatItems(allItems);
88              Map<BasketItem, Long> calcHighVat =
calculateHighVat(highVatItems);
89              long sumPriceWithVat = sumPrices(calcHighVat);
90
91              System.out.println(sumPriceWithVat);
92          }
93      }
```

*Ukázka kódu 44 Filter-Map-Reduce v jazyku Java*

V ukázce kódu 44 lze vidět zjednodušený příklad modelové situace. Řekněme, že v košíku elektronického obchodu jsou položky s cenami 1,00 – 5,00 (je dobrá praxe vyjádřit cenu celým číslem kvůli vyhnutí se nepřesnostem v reprezentaci desetinných čísel). Řekněme, že v daném státě se platí 20% DPH z položek, jejichž cena je 3 měnové jednotky a vyšší. V metodě `main` je vidět postup, kdy z vytvořeného košíku potřebujeme vytáhnout položky, u kterých se platí tato vysoká sazba DPH, pro každou položku potřebujeme spočítat cenu s DPH a následně ceny s DPH sečíst. V jazyku Java (do verze 8) je potřeba takovéto situace řešit vytvořením nové kolekce a zkopírováním příslušných dat do ní. Takovýto kód je relativně dlouhý a celá business logika v něm snadno zapadne.

```
01    case class BasketItem(price:Int)
02
03    object Main extends App{
04      def createBasket () = List(
05        new BasketItem(100),
06        new BasketItem(200),
07        new BasketItem(300),
08        new BasketItem(400),
09        new BasketItem(500)
10      )
11
12      val allItems = createBasket()
```

```
13      val highVatItems = allItems.filter(it => it.price >= 300)
14      val calcHighVat = highVatItems.map(it => (it,
math.round(it.price + (it.price * 0.2)))).toMap
15      val sumPriceWithVat = calcHighVat.map(it => it._2).reduce((p1,
p2) => p1 + p2)
16
17      print(sumPriceWithVat)
18    }
```

*Ukázka kódu 45 Filter-Map-Reduce v jazyku Scala*

V ukázce kódu 45 lze vidět kód vykonávající přesně to samé, jako kód v předchozí ukázce. První věc, které si lze všimnout, je třída `BasketItem`. Jedná se o case class, tedy typ třídy, která automaticky vygeneruje metody `hashCode`, `equals`, `toString`, getter/setter metody a konstruktor na základě parametrů. Samozřejmě lze do ní přidat další metody, ale zde to není nutné. Pokud bychom se podívali na bajtkód třídy `BasketItem` naimplementované v jazyku Java a Scala v těchto dvou ukázkách, byl by ekvivalentní.

Na kolekci vytvořených položek košíku je použita funkce `filter`. Ta přebírá jako svůj parametr funkci (může být i anonymní jako v této ukázce), často označovanou jako predikát, který jako parametr přebírá prvek kolekce a jehož návratová hodnota je typu `boolean`. Výstupem funkce `filter` je nová kolekce pouze s těmi prvky, pro které predikát nabývá pravdivé hodnoty.
V jazyku Scala je nová kolekce tzv. lazy, tedy se vyhodnocuje prvek po prvku až tehdy, kdy je potřeba. Důsledkem této vlastnosti je možnost práce s nekonečnými kolekcemi nebo výkonnostní výhoda, pokud není třeba přistoupit ke všem prvkům. Mezi hlavní nevýhody potom patří zpomalení operací, u kterých lze jinak očekávat, že budou rychlé, pokud přistupují ke kolekci, jejíž prvky jsou výsledkem náročných operací. Na následujícím řádku je vidět použití funkce `map`. Ta opět vrací lazy kolekci, nyní jsou však prvky nové kolekce výsledkem transformační funkce – která je předána jako parametr funkce `map`. Další funkcí v tomto příkladu je funkce `reduce`, která kolekci redukuje na 1 prvek. Tato funkce přebírá jako parametr funkci, jejíž vstupem jsou 2 prvky kolekce a výstupem výsledný prvek. V ukázkovém případě se jedná o součet dvou cen.

V této ukázce lze vidět kratší a stručnější kód než v ukázce 44. V případě, že programátor zná funkce filter, map a reduce, na první pohled vidí, co kód dělá, na rozdíl od ukázky v jazyku Java, kde se logická pravidla (výběr prvků s cenou vyšší než 3 měnové jednotky, výpočet ceny s DPH a součet cen s DPH) ztratí v množství kódu. Tento opakující se vzor je často nazýván jako **Filter-Map-Reduce** po funkcích, které se v tomto vzoru používají.

```
01    (defn create-basket
02      []
03      (list {:price 100}
04            {:price 200}
05            {:price 300}
06            {:price 400}
07            {:price 500}))
08
09    (defn add-high-vat
10      [price]
11      (+ price (* 0.2 price)))
```

```
12
13      (defn -main
14        [args]
15        (let [basket (create-basket)
16              high-vat-items (filter #(>= (:price %) 300) basket)
17              high-vats (map (fn [it] [it (add-high-vat (:price it))])
high-vat-items)
18              only-vats (map #(second %) high-vats)
19              vat-sum (reduce + only-vats)]
20          (print vat-sum "\n")))
```

*Ukázka kódu 46 Filter-Map-Reduce v jazyku Clojure*

V ukázce kódu 46 je vidět využití vzoru Filter-map-reduce v jazyku Clojure. Na rozdíl od jazyka Scala, kde tyto funkce jsou zpravidla volány nad objektem kolekce, v jazyku Clojure lze o globální funkce ze jmenného prostoru clojure.core. Tyto funkce zpravidla přebírají referenci na kolekci, kterou je třeba zpracovat, jako svůj poslední parametr. Predikát pro funkci `filter`, transformační funkce pro funkci `map` apod. jsou předávány jako první parametr.

Ve zmíněné ukázce lze vidět použití lambda výrazu pro funkci `filter` a druhý výskyt funkce `map`. Ten je v jazyce Clojure uvozen znakem #. Znak % pak znamená (jediný) parametr předaný výrazu (v případě použití více parametrů lze použít %1, %2 atd., kde %1 je první parametr – čísluje se od 1). Tam, kde lambda výraz nelze použít nebo je jeho použití nevhodné, lze použít anonymní funkci (první výskyt funkce `map` ve zmíněné ukázce) nebo přímo název funkce (jako u funkce `reduce` – zde byla předána funkce s názvem „+“).

### 1.5.2. Rekurze

Zatímco v imperativním programování je velmi rozšířeno použití cyklů, ve kterých se mění hodnoty proměnných, ve funkcionálním programování je obvyklejší použití rekurze bez změny hodnot. Některé jazyky, např. Clojure, ani klasické cykly jako *for* nebo *do* nenabízí. V dnešní době již téměř každý překladač funkcionálního jazyka nabízí optimalizaci koncové rekurze (anglicky tail-call elimination optimisation, dále jen TCO), kdy překladač je schopen koncovou rekurzi nahradit cyklem. Díky tomu lze eliminovat riziko přetečení zásobníku, které u klasické rekurze hrozí.

Typickým učebnicovým příkladem na použití rekurze je Fibonacciho posloupnost.

$$F(n) = \begin{cases} 0, & pro\ x = 0 \\ 1, & pro\ x = 1 \\ F(n-1) + F(n-2), & jinak \end{cases}$$

Přímým přepisem do jazyka Java bychom dostali kód, který lze vidět v ukázce 47.

```
01    public static int fibRec(int n) {
02        if (n == 0) return 0;
03        if (n == 1) return 1;
04
05        return fibRec(n - 1) + fibRec(n - 2);
06    }
```

*Ukázka kódu 47 Fibonacciho posloupnost jako rekurze v jazyku Java*

Takový kód, ač snadno čitelný a vracející správné výsledky, bohužel není dobrý. Pro každé volání funkce *fibRec* se opět zavolá funkce *fibRec*, a to hned 2x. Přetečení zásobníku tak nastane pro relativně malé hodnoty vstupního čísla *n*. Tento kód ani není efektivní, protože se 1 konkrétní hodnota může počítat několikrát. Např. pro n = 8 se hodnota pro 4 bude počítat celkem 5x. A ani kdyby překladač jazyka podporoval TCO, na tento kód by to nemělo žádný efekt. Tato optimalizace vyžaduje, aby poslední výraz ve funkci bylo buď navrácení konkrétní hodnoty, nebo volání této funkce. To zde splněno není, neboť posledním výrazem ve funkci *fibRec* je součet.

```
01    public static int fibLoop(int n) {
02        if (n == 0) return 0;
03        if (n == 1) return 1;
04
05        int prev = 1, prevPrev = 0, result = 0;
06        for (int i = 1; i < n; i++) {
07            result = prev + prevPrev;
08            prevPrev = prev;
09            prev = result;
10        }
11
12        return result;
13    }
```

*Ukázka kódu 48 Fibonacciho posloupnost jako cyklus v jazyku Java*

V ukázce kódu 48 lze vidět přepis funkce pro výpočet Fibonacciho posloupnosti jako cyklus. Tento kód již není tak snadno čitelný jako rekurzivní zápis, na druhou stranu je mnohem efektivnější – hodnota pro 1 konkrétní vstup je počítána pouze 1x a nehrozí přetečení zásobníku.

```
01    def fibTco(n : Int) : Int = {
02      @tailrec
03      def loop(idx : Int, acc : Int, prev : Int) : Int = idx match {
04        case 1 => acc
05        case _ => loop(idx - 1, acc + prev, acc)
06      }
07
08      n match {
09        case 0 => 0
10        case 1 => 1
11        case _ => loop(n, 1, 0)
12      }
13    }
```

*Ukázka kódu 49 Fibonacciho posloupnost jako rekurze v jazyku Scala*

V ukázce kódu 49 lze vidět přepis kódu z ukázky 48 do podoby, která v jazyku Scala dokáže využít TCO. Je zde vidět funkce *fibTCO*, v jejímž těle je obsažena vnitřní funkce *loop* a poté rozhodovací struktura, která dle vstupní hodnoty *n* přímo vrátí konkrétní číslo, nebo spustí výpočet. K označení funkce, u které by měl překladač použít TCO, se používá anotace *@tailrec* – zde byla tato anotace použita u vnitřní funkce *loop*. Tato funkce provádí v podstatě ty samé kroky,

jako for cyklus z ukázky 48. Kód je však kratší a dle mého názoru čitelnější. Překladač jazyka Scala, díky anotaci *@tailrec,* vnitřní funkci *loop* buď přeloží jako cyklus, nebo vyvolá chybu při překladu [15]. Autor kódu si díky tomu může být jist, že k optimalizaci skutečně dojde a nebude jej čekat nemilé překvapení v podobě přetečení zásobníku.

```
01    (defn fib
02      [n]
03      (case n
04        0 0
05        1 1
06        (loop [idx n
07               acc 1
08               pre 0]
09          (if (= idx 1)
10            acc
11            (recur (- idx 1)
12                   (+ acc pre)
13                   acc)))))
```

*Ukázka kódu 50 Fibonacciho posloupnost jako rekurze v jazyku Clojure*

V ukázce kódu 50 lze vidět ten samý kód v jazyku Clojure. Překladač jazyka Clojure nepodporuje TCO, proto je v tomto kódu třeba použít konstrukce *loop – recur*[11]. Konstrukce *loop* obsahuje počáteční vazby hodnot, které jsou při následné iteraci přepsány parametry výrazu *recur*. Cyklus se bude opakovat, dokud výsledek vyhodnocení jeho těla bude výraz *recur*. Pokud bude výsledkem konkrétní hodnota, cyklus skončí a výsledkem cyklu bude právě tato hodnota, podobně jako u rekurzivního volání funkcí. Kombinace těchto konstrukcí zajistí stejný efekt, jako anotace *@tailrec* v jazyku Scala.

V době psaní této práce bylo pro programy v jazyku Clojure běžné, že ve výsledných JAR/WAR archivech byly distribuovány ve formě zdrojových kódů a zdrojové kódy byly překládány až v případě potřeby. Pro takovou situaci je žádoucí, aby překlad probíhal co nejrychleji a počet kontrol, které překladač provádí, byl co nejmenší. Fakt, že se v případě *loop* - *recur* se nejedná o rekurzivní volání funkcí, ale o speciální konstrukci, automaticky autorovi dává jistotu, že nedojde k přetečení zásobníku i bez speciální kontroly při překladu programu. Naopak překladač jazyka Scala je pověstný důkladnou kontrolou co nejvíce věcí (a v době psaní práce byl díky tomu překlad programů napsaných v jazyku Scala relativně pomalý), proto pravděpodobně autoři jazyka Scala zvolili čitelnější a hezčí formu zápisu v kombinaci s další kontrolou při překladu.

### 1.5.3. Vytváření funkcí

Ve funkcionálním programování lze vidět zaběhlý postup, kdy funkce, na základě svých parametrů, vrací jinou, často nově vytvořenou anonymní funkci. Využívá se zde té vlastnosti funkcionálních jazyků, že funkce jsou plnohodnotnými prvky a lze je tedy vytvářet za běhu,

[11] Více dokumentace ke speciálním formám *loop* a *recur* lze nalézt např. na stránkách clojuredoc.org.
http://clojuredocs.org/clojure.core/loop
http://clojuredocs.org/clojure.core/recur

předávat jako argumenty jiných funkcí, vracet je z funkcí nebo částečně aplikovat. Tento vzor lze opět ukázat přímo na praktickém příkladu.

V ukázce kódu č. 44 na stránce 48 lze vidět fiktivní výpočet ceny zboží, kdy z určitých položek se platí DPH. Ve funkcionálním jazyku lze tento příklad modifikovat. Bude existovat jedna funkce, která spočítá DPH pro určité zboží. Samotné tělo výpočtové funkce by záleželo na konkrétním regionu, ve kterém je třeba zboží danit. Můžeme tedy chtít funkci, jejímž vstupem bude region a výstupem bude funkce na výpočet DPH. V imperativním programování by se takový problém pravděpodobně řešil použitím návrhového vzoru strategy nebo command.

```
01      abstract class Country
02
03      case object Country1 extends Country
04      case object Country2 extends Country
05
06      def getVatFunc(region : Country) : (Int => Int) = {
07        region match {
08          case Country1 => (price : Int) =>
09            if (price < 300) math.round((0.1f * price) + price)
10            else math.round((0.2f * price) + price)
11          case Country2 => (price : Int) =>
12            if (price < 400) math.round((0.15f * price) + price)
13            else math.round((0.25f * price) + price)
14        }
15      }
```

*Ukázka kódu 51 Vytvoření anonymní funkce v jazyku Scala*

V ukázce 51 lze vidět funkci *getVatFunc*, která vrátí funkci pro výpočet DPH pro daný region. V hlavičce funkce je uvedena návratová hodnota jako *(Int => Int)*, tedy funkce s parametrem typu *Int* a návratovou hodnotou *Int*. Na základě vstupního parametru, země, je vytvořena funkce pro výpočet DPH. Takto získanou funkci lze použít např. jako vstup funkce *map* při procházení kolekce se zbožím v košíku a výpočtu cen. Díky tomu, že se při výpočtu ceny konkrétního zboží v tomto případě ušetří rozhodování, který algoritmus použít (neb je pro všechny položky stejný) a jak jej sestavit (jedná-li se o komplexní algoritmus založený např. na uživatelem nastavených pravidlech), může se tímto jednoduchým způsobem významně zkrátit výpočet celkové ceny.

V jazyku Clojure (Ukázka kódu 52) lze dostat velmi podobný kód.

```
01    (defn get-vat-func
02      [country]
03      (case country
04      :country1 (fn [price] (if (< price 300)
05                              (Math/round (+ price (* 0.1 price)))
06                              (Math/round (+ price (* 0.2 price)))))
07      :country2 (fn [price] (if (< price 400)
08                              (Math/round (+ price (* 0.15 price)))
09                              (Math/round (+ price (* 0.25
price))))))))
```

*Ukázka kódu 52 Vytvoření anonymní funkce v jazyku Clojure*

# 3. Paralelismus

Téměř všechny široce používané programovací jazyky či prostředí poskytují prostředky, díky kterým lze vytvářet programy, jejichž kód je prováděn paralelně. V prostředí Java se jedná především o vlákna a zámky. Tento model umožňuje přesnou vládu nad děním, ale bohužel je náchylný na chyby způsobující situace jako uváznutí (deadlock) nebo souběh (race condition). Bez správného ošetření vedou tyto situace k selhání systému. Najít takovou chybu bývá často velmi těžké, protože vznik sekvence událostí, které umožní projevení dané chyby, je často dílem náhody. Chyba se tedy nemusí projevit v prostředí, kde jsou dostupné vývojové nástroje a její odstranění je tak velmi náročné.

Jak je uvedeno v úvodu práce, v budoucnu lze zřejmě očekávat zvyšující se zájem o software, který umožní využít možností víceprocesorových počítačů. To s sebou ponese vyšší tlak na paralelní programování a tedy i zvýšené riziko selhání systémů díky chybám v této oblasti. Dobrým příkladem takového tlaku je dle mého názoru systém Windows Phone 8, kde Microsoft mnohdy programátorům nedává na výběr, zdali použijí synchronní či asynchronní API. Součástí Microsoft .Net frameworku je i třída *System.Net.HttpWebRequest*[12], která slouží k reprezentaci http požadavku. Ve verzi Microsoft .Net frameworku určeném pro osobní počítače lze využít jak její synchronní funkce (např. *GetResponse*), které při svém vykonávání zablokují právě běžící vlákno a mohou tedy pozastavit běh celého programu, tak i jejich asynchronní verze (např. *GetResponseAsync*), které právě běžící vlákno nepozastaví. Nicméně ve verzi Microsoft .Net frameworku určeném pro tablety a mobilní telefony již synchronní API podporováno není a programátoři jsou tedy nuceni k využití asynchronního API a potýkání se s problémy paralelního běhu různých částí programu[13].

Jazyky Clojure a Scala již od začátku byly navrhovány s myšlenkou na paralelní programování. Jazyk Clojure má zabudovanou podporu pro využití modelu *Software transactional memory* a součástí základní knihovny jazyka Scala byla již od počátku podpora *Actor modelu* (později byla tato podpora vyčleněna jako samostatný *Akka* framework[14], který je dostupný i pro jazyk Java). Tyto dva modely budou podrobněji rozebrány v této kapitole.

### 1.5.4. Actor model

Actor model je matematický model paralelismu, který byl původně vytvořen pro výpočty v oblasti umělé inteligence. Jeho základem je nezávislý prvek s definovaným chováním, *actor,* který reaguje na zprávy zvenčí (od jiných actorů) a to buď změnou svého stavu, změnou vzorce chování, nebo zasláním zprávy. [16] Zprávy, které nelze zpracovat ihned, čekají ve schránce na zpracování, jsou zpracovávány sériově a v základním modelu uvedeném v [16] není jejich pořadí definováno (tedy mohou být zpracovávány v libovolném pořadí). Každý actor je tedy samostatnou nezávislou jednotkou a běží nezávisle na ostatních actorech, paralelně ve stejném čase. Tuto nezávislost uvádějí autoři článku [16] jako hlavní rozdíl mezi actor modelem a objektově orientovaným

[12] Dokumentace k této třídě je k dispozici na webu msdn.microsoft.com http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest%28v=vs.110%29.aspx

[13] Aby moje poznámka nevyzněla nespravedlivě, hodí se říci, že do jazyka C# verze 5.0 zároveň přibyla syntaktická podpora pro jednoduché asynchronní programování – hojně propagovaná klíčová slova *async* a *await*. Toto obvykle stačí na úlohy, kdy program čeká na vstupně-výstupní operace.

[14] Více informací o tomto frameworku lze nalézt na oficiálních stránkách http://akka.io

programováním. V objektově orientovaném programování též objekty komunikují zasíláním zpráv a jejich chování může být závislé na jejich vnitřním stavu a přijaté zprávě, ale kód programu napsaném v objektově orientovaném jazyce je zpravidla zpracováván sériově [16].

Jako příklad jednoduchého actoru je v [16] uveden typ *cell* (buňka). Jedná se o actora, který má uloženu nějakou hodnotu a rozumí dvěma typům zpráv – na získání hodnoty a na změnu hodnoty.

```
01    public class Cell {
02        private int state;
03
04        public Cell(int startingState) {
05            state = startingState;
06        }
07
08        public synchronized int getState() {
09            return state;
10        }
11
12        public synchronized void setState(int state) {
13            this.state = state;
14        }
15    }
```

*Ukázka kódu 53 Cell v jazyku Java*

Pro názornost lze uvést, jak by mohla vypadat implementace prvku cell v jazyku Java, viz ukázka 53. Díky použití klíčového slova *synchronized* je zajištěno, že ačkoliv si přečtení/změnu hodnoty buňky může vyžádat více objektů z více vláken, konkrétní instance buňky bude v každém bodě v čase reagovat pouze na 1 požadavek. Ostatní budou čekat na zpracování a nedojde ke vzájemné kolizi.

Ukázka kódu 54 představuje implementaci prvku Cell v jazyku Scala s použitím Akka frameworku. Přestože tento framework lze využít i v jazyku Java, využití jazyka Scala je v tomto případě výhodnější díky bohatší syntaxi a širším možnostem, což vede ke kratšímu a expresivnějšímu kódu.

Na začátku kódu si lze povšimnout definic zpráv. Zpráva *GetValue* je definována jako *case object*, existuje tedy jediná instance této zprávy (v klasickém objektově orientovaném jazyku by pravděpodobně byl v tomto případě použit návrhový vzor Singleton), neboť pokaždé, kdy bude tato zpráva zaslána, bude stejná jako v ostatních případech. Zprávy *SetValue* a *Value* jsou již jako třídy a budou zasílány výhradně instance těchto tříd, neboť tyto zprávy při každém použití budou mít jiné tělo (zde hodnotu). Zprávy *GetValue* a *SetValue* jsou určeny k odeslání do actoru *Cell*, *Value* je zpráva, kterou bude actor posílat zpět.

Na řádku 05 začíná definice třídy *Cell*, která je rozšířením třídy *Actor*. Lze v ní vidět 1 proměnnou pro uchování stavu a 1 metodu, která představuje chování actora. Dle druhu přijaté zprávy actor buď změní svůj stav, nebo odesílateli zprávy pošle zprávu obsahující hodnotu jeho stavu. Pro zaslání zprávy je zde použit operátor „!“, reference na odesílatele zprávy je v hodnotě *sender*.

Řádek 15 je začátek hlavní metody příkladu. Nejprve je vytvořen systém, jehož součástí budou jednotlivé actory. Systémy jsou nezávislé na sobě a na vnějším okolí, v jedné aplikaci může existovat více systémů a systémy mohou být distribuované. Dále je vytvořena instance actoru, *cell*. Instance actorů se nevytváří přímo, ale používá se zde factory metoda systému, do kterého má být instance actoru zařazena. Nakonec je vytvořena schránka, která slouží pro komunikaci mezi vnějším světem a systémem actorů.

```
01    case object GetValue
02    case class SetValue(value : Int)
03    case class Value(value : Int)
04
05    class Cell extends Actor {
06      var value : Int = 0
07
08      def receive = {
09        case GetValue => sender ! Value(value)
10        case SetValue(newVal) => value = newVal
11      }
12    }
13
14    object CellExample {
15      def main(args: Array[String]): Unit = {
16        val system = ActorSystem("cellExample")
17        val cell = system.actorOf(Props[Cell])
18        val inbox = Inbox.create(system)
19
20        cell.tell(SetValue(10), Actor.noSender)
21        inbox.send(cell, GetValue)
22        val Value(val1) = inbox.receive(1.seconds)
23        println(s"Value 1: $val1")
24
25        cell.tell(SetValue(15), Actor.noSender)
26        inbox.send(cell, GetValue)
27        val Value(val2) = inbox.receive(1.seconds)
28        println(s"Value 2: $val2")
29
30        system.shutdown()
31        system.awaitTermination()
32      }
33    }
```

*Ukázka kódu 54 Cell v jazyku Scala s využitím Akka frameworku*

V následujícím bloku, začínajícím řádkem 20 , je actoru *cell* poslána zpráva pro nastavení hodnoty. Pro poslání této zprávy bylo využito volání metody *tell*, které je shodné s použitím operátoru „*!*“. Tato zpráva je poslána anonymně, bez odesílatele (viz druhý parametr volání funkce), a není možno na ni zachytit reakci. Následuje opět odeslání zprávy actoru *cell*, tentokrát se ale pro odeslání nepoužívá přímo konstrukce „příjemce ! zpráva“ – jelikož tento kód běží vně

systému actorů, nebylo by možné doručit zpět odpověď. Proto je využit objekt schránky. V tomto případě byla pro odeslání zprávy použita funkce *send* (která je ekvivalentní s operátorem „*?*“) a očekává se odpověď. Ta v tomto případě dorazí do schránky, odkud je možno si ji vyzvednout pomocí metody *receive* a určitý maximální čas na její doručení čekat. Hodnota vytištěná na standardní výstup v řádku 23 bude „*Value 1: 10*“ – hodnota, na kterou buňka změnila svůj stav po obdržení zprávy odeslané v řádku 20 . Na konec je dobré ještě korektně ukončit systém.

Rozhodnutí sdružit actory do systému umožnilo přidat koncept dozoru (supervision). Každý actor v systému má rodiče, který odchytává neošetřené výjimky, které v průběhu vykonávání jeho kódu vzniknou. V případech, kdy v podřízeném actoru nastane neošetřená výjimka, může nadřazený actor (supervisor) reagovat buď zahozením výjimky (*Resume*), restartem podřízeného od začátku bez uchování jeho stavu (*Restart*), jeho zastavením (*Stop*) nebo eskalovat výjimku výše (*Escalate*). Tyto možné reakce (strategie) lze aplikovat buď na jednoho konkrétního podřízeného actora, ve kterém výjimka nastala (*OneForOneStrategy*), nebo na všechny podřízené (*OneForAllStrategy*).

```
01    class Supervisor extends Actor {
02
03      override val supervisorStrategy =
04        OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1
minute) {
05          case _: ArithmeticException => Resume
06          case _: NullPointerException => Restart
07          case _: IllegalArgumentException => Stop
08          case _: Exception => Escalate
09      }
10
11    ...
12
13    }
```

*Ukázka kódu 55 Nastavení supervisorStrategy v nadřízeném actoru. Převzato z [18].*

V ukázce kódu 55 lze vidět nastavení strategie, jak reagovat na výjimky z podřízených actorů. Je zde využita *OneForOneStrategy*, tedy reakce se bude týkat pouze konkrétního actora, při jehož běhu neošetřená výjimka nastala. Následuje seznam výjimek, na které budou následovat různé reakce a nakonec výjimka typu *Exception* (tedy všechny ostatní výjimky vyjma uvedených výše, neboť třídy pro výjimky dědí od třídy *Exception*), která bude eskalovaná výše.

### 1.5.5. Software transactional memory

Koncept Software transactional memory (STM) vznikl jako návrh na vyřešení nedostatku funkcionálních programovacích jazyků. Existují algoritmy, pro které je čistě funkcionální přístup nevhodný. Pro tvůrce algoritmů je mnohem snazší přemýšlet nad „vnitřním měnícím se stavem“ než v čistě funkcionálním, neměnném prostředí. Přestože oba přístupy jsou univerzální a tedy všechny algoritmy mohou být implementovány ve funkcionálních i imperativních jazycích, programátoři zpravidla považují imperativní přístup za jednodušší na přemýšlení, design i opravu chyb. [19]

Proto Tom Knight ve svém článku [19] navrhuje podobný přístup ke správě změnitelného stavu v paměti, jaký bývá implementován v databázích – transakce. Vychází z předpokladu, že program lze rozdělit na bloky kódu tak, že každý blok bude mít čistě funkcionální část (s možností čtení dat ze sdíleného místa v paměti) a část s postranními efekty (s možností zápisu dat). Část s postranními efekty bude vždy na konci bloku. Zároveň těsně před provedením zápisu navrhuje zjistit, zdali nějaká jiná transakce neprovedla změnu hodnoty, která byla v průběhu prováděného bloku přečtena. Každý takový blok zde přestavuje transakci, kde je zaručena atomicita, konzistence a izolovanost[15].

Díky těmto vlastnostem je zaručeno, že nebude docházet k nepříjemným situacím jako je deadlock nebo race condition, které jsou při explicitním paralelním programování s využitím vláken a zámků celkem častým jevem. Daní za to je skutečnost, že některé transakce selžou a bude třeba je celé opakovat i v případě, který by při použití vláken a zámků proběhl v pořádku a dal správný výsledek. Pokud v těle takové transakce bude výpočetně náročná operace, může tento fakt způsobit citelné snížení výkonu programu. Stejně tak je třeba zajistit, že uvnitř transakce skutečně nebude docházet k nechtěným postranním efektům (např. výstup do konzole nebo odpálení rakety), neboť opakování transakce by mělo za následek i opakování daného postranního efektu.

Základní knihovna jazyka clojure již má podporu pro STM přímo v sobě (při využití STM v jazyku scala je třeba využít nějakou další knihovnu či framework – např. výše zmíněný Akka). Měnitelné reference na objekty jsou v jazyku clojure označovány jako *ref* a jejich hodnota může být změněna pomocí funkcí *ref-set* nebo *alter* a tyto změny je třeba provést bloku *dosync* [20].

Funkce *ref* slouží k vytvoření měnitelné reference (funkce *deref* nebo @ vrátí hodnotu reference). Při vytváření reference lze také určit validační funkci reference. Funkce *ref-set* nastaví novou hodnotu reference a *alter* změní hodnotu reference (pomocí předané transformační funkce). *Dosync* vytvoří transakční blok. Více informací lze získat z dokumentace k jazyku clojure, viz [21].

Oproti návrhu v článku [19] zde není nutné měnit hodnoty referencí až na konci transakčního bloku, funkce *ref-set* a *alter* lze volat i v jeho průběhu a tělo transakčního bloku může obsahovat postranní efekty. Je ale třeba mít na paměti, že v případě selhání transakce je běh bloku přerušen a ne všechny postranní efekty tedy proběhnou a při opakování transakce všechny proběhnou znovu. Této vlastnosti je možno využít například v průběhu ladění programu, kdy je třeba zjistit, zdali nějaká transakce neselhává často a netvoří tedy úzké hrdlo. Pokud se v průběhu transakce změní hodnota reference, další část transakce může pracovat s touto změněnou hodnotou (nicméně pravidlo izolovanosti porušeno není, ostatní transakční bloky pracují s původní hodnotou).

```
01    (import '(java.util.concurrent Executors))
02
03    (defn test-stm [nitems nthreads niters]
04      (let [refs  (map ref (repeat nitems 0))
05            pool  (Executors/newFixedThreadPool nthreads)
06            tasks (map (fn [t]
07                         (fn []
08                           (dotimes [n niters]
```

[15] Trvanlivost nikoliv, neboť se v tomto modelu zpravidla nejedná o data na trvalém úložišti, ale v operační paměti.

```
09                                (dosync
10                                  (doseq [r refs]
11                                    (alter r + 1 t))))))
12                        (range nthreads))]
13        (doseq [future (.invokeAll pool tasks)]
14          (.get future))
15        (.shutdown pool)
16        (map deref refs)))
17
18    (test-stm 10 10 10000)
19    -> (550000 550000 550000 550000 550000 550000 550000 550000 550000
550000)
```

*Ukázka kódu 56 Příklad použití Ref v jazyku clojure. Převzato z [20]*

Ukázka kódu 56 představuje příklad využití STM. Každá z 10 *ref* hodnot je celé číslo, počáteční hodnota je nastavena na 0 (řádek 04 ). Dále je vytvořen *FixedThreadPool*, množina vláken. Přestože lze využít i jiné možnosti paralelismu, než vlákna z prostředí Java, autor této ukázky kódu a jazyka clojure, Rich Hickey, chtěl poukázat na to, že tato možnost existuje (byť se v programech napsaných v jazyku clojure doporučuje využití systému agentů[16]). Každé z vláken se snaží 10000x inkrementovat všechny *ref* hodnoty, každá inkrementace má vlastní transakci. Jak je vidět na výsledném stavu programu (poslední řádek), všechny *ref* hodnoty jsou stejné – program skončil korektně.

[16] Více o tomto systému, který je opět součástí základní knihovny jazyka clojure, se lze dozvědět z dokumentace dostupné z http://clojure.org/agents

# 4. Integrace

Tato kapitola popisuje způsoby, jak integrovat moduly vytvořené ve funkcionálních a objektově orientovaných jazycích a jaké problémy u toho mohou nastat. Ne vždy existuje možnost vytvořit celý program znovu od začátku v jazyku dle vlastní volby. Požadavky na integraci nové funkcionality do stávajícího software jsou častým jevem, neboť životní cyklus některých aplikací může být velmi dlouhý. Například historie dobře známého textového editoru Emacs se začala psát v roce 1976[17] a ze stejné doby pochází také editor Vi, který je nyní častou součástí operačních systémů založených na Linux nebo UNIX jádrech. Pro oba dva editory existuje množství rozšíření, z nichž mnohé jsou stále v aktivním vývoji[18], stejně jako zmíněné editory.

Nejprve budou popsány způsoby integrace modulů vytvořených v programovacím jazyku Java do modulů v programovacích jazycích Scala a Clojure a vytvoření modulů pomocí kombinace programovacích jazyku Scala+Java a Clojure+Java. Následně bude popsán opačný postup, integrace modulů vytvořených v programovacích jazycích Scala/Clojure do modulů vytvořených v jazyku Java. Pro poslední zmíněný případ bude uvedena komplexnější praktická ukázka.

## 1.6. Integrace Java modulů

Díky tomu, že jazyky Clojure a Scala primárně cílí na Java Virtual Machine, je možno z programů v nich vytvořených využít jak základní knihovny jazyka Java, tak knihoven přeložených do Java bajtkódu. Díky tomu lze eliminovat typický neduh nově vznikajících či méně rozšířených programovacích jazyků – nízký počet použitelných knihoven.

Přestože pro sestavení programů vytvořených v jazycích Scala nebo Clojure lze použít příkazovou řádku, obecně bývá výhodnější využít build systém. Pro projekty vytvořené v jazyku Scala bývá často využíván systém SBT[19] a pro projekty vytvořené v jazyku Clojure bývá využíván systém Leiningen[20]. Další populární volbou bývá systém Maven, který je však dobře známý a není třeba jej představovat.

Pro jednoduchý avšak praktický příklad lze využít knihovnu Apache Commons Exec, která poskytuje jednoduché API pro spouštění externích příkazů a čtení výstupů. Oba dva systémy, jak SBT tak Leiningen, používají repozitáře určené pro systém Maven.

```
01    SBT:
02
03    libraryDependencies += "org.apache.commons" %% "commons-exec" %
"1.3"
04
05
06    Leiningen:
07
08    :dependencies [[org.apache.commons/commons-exec "1.3"]]
```

[17] Více informací viz oficiální stránky projektu GNU Emacs http://www.gnu.org/software/emacs/
[18] Například CIDER (https://github.com/clojure-emacs/cider), rozšíření pro GNU Emacs, bylo v době vytváření této práce v aktivním vývoji a projekt měl přibližně 100 přispěvatelů.
[19] Více informací o tomto projektu lze najít na oficiálních webových stránkách: http://www.scala-sbt.org/
[20] Více informací o tomto projektu lze najít na oficiálních webových stránkách: http://leiningen.org/

```
09
10    Maven:
11
12    <dependency>
13      <groupId>org.apache.commons</groupId>
14      <artifactId>commons-exec</artifactId>
15      <version>1.3</version>
16    </dependency>
```
*Ukázka kódu 57 Definice závislostí na externí knihovně v systémech SBT, Leningen a Maven*

V ukázce kódu 57 lze vidět různé zápisy definice závislosti na knihovně Apache Commons Exec. Všechny mají stejný význam – při sestavování projektu se daná knihovna včetně závislostí automaticky stáhne z repozitáře (ve výchozím nastavení Maven Central, nicméně lze využít jakýkoliv jiný – např. soukromý firemní či projektový repozitář) a přidá do classpath současného projektu.

Jazyk Scala nemá rozdílnou syntaxi pro práci s třídami vytvořenými v jazyku Scala a Java. Proto s třídami z této knihovny lze pracovat stejně, jako by se jednalo o třídy jazyka Scala.

```
01    import org.apache.commons.exec.{CommandLine, PumpStreamHandler,
DefaultExecutor}

02    object ExecTest extends App {
03      val exec = new DefaultExecutor()
04      exec.setStreamHandler(new PumpStreamHandler())
05      val cmdLine = CommandLine.parse("cmd /c dir .")
06      exec.execute(cmdLine)
07    }
```
*Ukázka kódu 58 Příklad využití externí knihovny z jazyku Scala*

V jazyku Clojure je třeba pro práci s třídami využít specifickou syntaxi. Pro volání statických metod se jedná o „název_třídy/název_metody“, pro volání konstruktoru „název_třídy.“ (tedy název třídy a znak tečka) a pro volání metody na instanci buď „.. instance název_metody parametry metody“ či „.název_metody instance parametry metody“ (tedy znak tečka bezprostředně následován názvem metody). Druhý uvedený způsob je častější.

```
01    (ns external-lib-clojure.core
02      (:import [org.apache.commons.exec
03                  DefaultExecutor
04                  CommandLine
05                  PumpStreamHandler])
06      (:gen-class))
07
08    (defn run-ext-cmd []
09      (let [exec (DefaultExecutor.)
10            _    (.setStreamHandler exec (PumpStreamHandler.))
11            cmd  (CommandLine/parse "cmd /c dir .")]
12        (.execute exec cmd)))
```

```
13
14    (defn -main
15      [& args]
16      (run-ext-cmd))
```
*Ukázka kódu 59 Příklad využití externí knihovny z jazyku Clojure*

V ukázkách výše si lze všimnout, že v případě jazyka Scala není definována žádná metoda main, neboť tělo objektu *ExecTest* slouží jako hlavní metoda aplikace. Naproti tomu v jazyku Clojure je metoda main definována a pouze zavolá metodu *run-ext-cmd*, která zavolá externí příkaz. Za zmínku stojí řádek 10 ukázky kódu 59. Návratová hodnota metody *setStreamHandler* se zdánlivě uchovává jako hodnota s návem „_“, nicméně znak podtržítka v tomto kontextu znamená, že se návratová hodnota nikam ukládat nemá. Funkce *setStreamHandler* je volána čistě kvůli postrannímu efektu.

Programy z těchto dvou ukázek, pokud by byly spuštěny v operačním systému Windows, načtou obsah aktuálního pracovního adresáře (příkaz „*dir .“*) a zobrazí jej na standardním výstupu (o to se stará třída *PumpStreamHandler*, která „pumpuje“ standardní vstup programu do spuštěného externího procesu a standardní výstup a standardní chybový výstup externího procesu přesměrovává do standardního výstupu a standardního chybového výstupu hlavního programu).

Zatímco využití této knihovny z jazyku Scala vypadalo velice podobně jako by byla využita z jazyka Java, u jazyku Clojure vypadalo použití velmi odlišně. Při využití jazyku Clojure je ne vždy zcela zjevné či pohodlné využití knihoven jazyka Java. Proto může být dobrou volbou vytvořit kombinovaný Clojure+Java projekt a externí knihovny, jejichž použití přímo z jazyka Clojure by mohlo být složité, použít z jazyka Java a vytvořit dobře použitelné rozhraní.

V případě využití systému Leiningen stačí v projektovém souboru nastavit cestu ke zdrojovým souborům jazyka Java (pomocí klíče *java-source-paths*) a následně lze mít část projektu v jazyku Clojure a část v jazyku Java.

```
01      :source-paths ["src/main/clojure"]
02      :java-source-paths ["src/main/java"]
```
*Ukázka kódu 60 Projektový soubor systému Leiningen kombinovaného Clojure + Java projektu*

Při použití systému SBT stačí java soubory umístit do adresáře „*src/main/java“*, není třeba měnit projektový soubor.

U kombinovaných projektů Clojure+Java lze narazit na obtíž vyplývající z dynamické povahy jazyka Clojure. Při sestavování zdrojových souborů jazyka Java je nutné, aby všechny třídy, na kterých kód závisí, byly zkompilované a přístupné. Proto, v případě, že kód vytvářený v jazyku Java závisí na kódu vytvářeném v jazyku Clojure, je nutné, aby daný jmenný prostor jazyka Clojure byl staticky kompilovaný dopředu. Tato skutečnost se dá zajistit tak, že se do definice jmenného prostoru přidá klíč „:gen-class“ a v projektovém souboru se název jmenného prostoru přidá do vektoru pod klíčem „:aot“. Tímto se zajistí přístupnost Clojure kódu pro Java kód. Stejná podmínka platí i pro jmenný prostor, ve kterém se nachází funkce main. Kombinované programy v jazycích Scala+Java tímto neduhem netrpí, neboť programy v jazyku Scala jsou kompilované staticky a překladač jazyka Scala umí zjistit a uspokojit závislosti mezi Java a Scala třídami.

## 1.7. Integrace Scala a Clojure modulů

Do stávajících programů (např. vytvořených v jazyku Java) lze integrovat moduly vytvořené v jazycích Scala a Clojure. Programy v jazyku Scala jsou překládány do stejného bajtkódu Java Virtual Machine jako programy vytvořené v jazyku Java a lze je typicky ihned použít. Programy v jazyku Clojure lze integrovat dvěma způsoby. Buď lze integrovat přímo běhové prostředí jazyka a programy v jazyku Clojure spouštět jako skripty (s využitím Just-In-Time kompilace), což může být žádoucí, pokud je třeba uživatelům systému nabídnout možnost systém rozšiřovat o pluginy či vlastní logická pravidla. Druhá možnost je staticky zkompilovat Clojure program do Java bajtkódu, čímž ze jmenných prostorů vzniknou třídy, které lze používat jako jakékoliv jiné třídy, podobně jako je tomu v případě jazyka Scala.

Programy vytvořené v jazycích Scala a Clojure potřebují (nad rámec standardní knihovny jazyka Java) ještě vlastní běhová prostředí. U každého se jedná jeden jar balíček. V době vytváření této práce byly aktuální verze jazyka Scala 2.11.5 a Clojure 1.6.0. Velikost běhového prostředí jazyka Scala této verze je cca 5,5MB a Clojure 3,6MB (lze použít i „slim“ verzi, která má velikost 1MB, ale neobsahuje předkompilovaný kód, tedy celé běhové prostředí se při spuštění programu kompiluje, což prodlužuje čas potřebný ke spuštění programu).

Základní knihovna jazyka Scala definuje mimo jiného i vlastní třídy pro kolekce – od každé existují měnitelné a neměnné varianty. Tyto kolekce bohužel nejsou kompatibilní s Java kolekcemi. Proto je žádoucí při přebírání dat z Java modulů tyto kolekce převést na Scala varianty, neboť mnoho funkcionálních operací nelze v jazyku Scala provést nad Java kolekcemi. Při předávání dat ze Scala modulu do Java modulu je vhodné kolekce převést na Java varianty, neboť rozhraní těchto kolekcí může být z jazyka Java těžko použitelné. K těmto převodům lze využít jmenných prostorů *scala.collections.JavaConversions* a *scala.collections.JavaConverters*. Zde jsou definovány třídy a operace pro explicitní i implicitní převody.

Kolekce jazyka Clojure implementují rozhraní *java.util.Collection*, díky čemuž lze kolekce jazyka Clojure přímo použít v jazyku Java bez větších obtíží. Stejně tak Java kolekce lze dobře využít z jazyka Clojure. Nicméně v případě, že je třeba předat kolekci dat do Java modulu, je lepší využít Java kolekce. Převod je opět jednoduchý. V případě převedení Java kolekce na Clojure kolekci lze využít funkci *into*, která spojí dvě kolekce. V případě převedení Clojure kolekce na Java variantu lze využít přímo konstruktor Java kolekcí (či map).

```
01    (into [] (java.util.ArrayList. [1 2 3]))
```

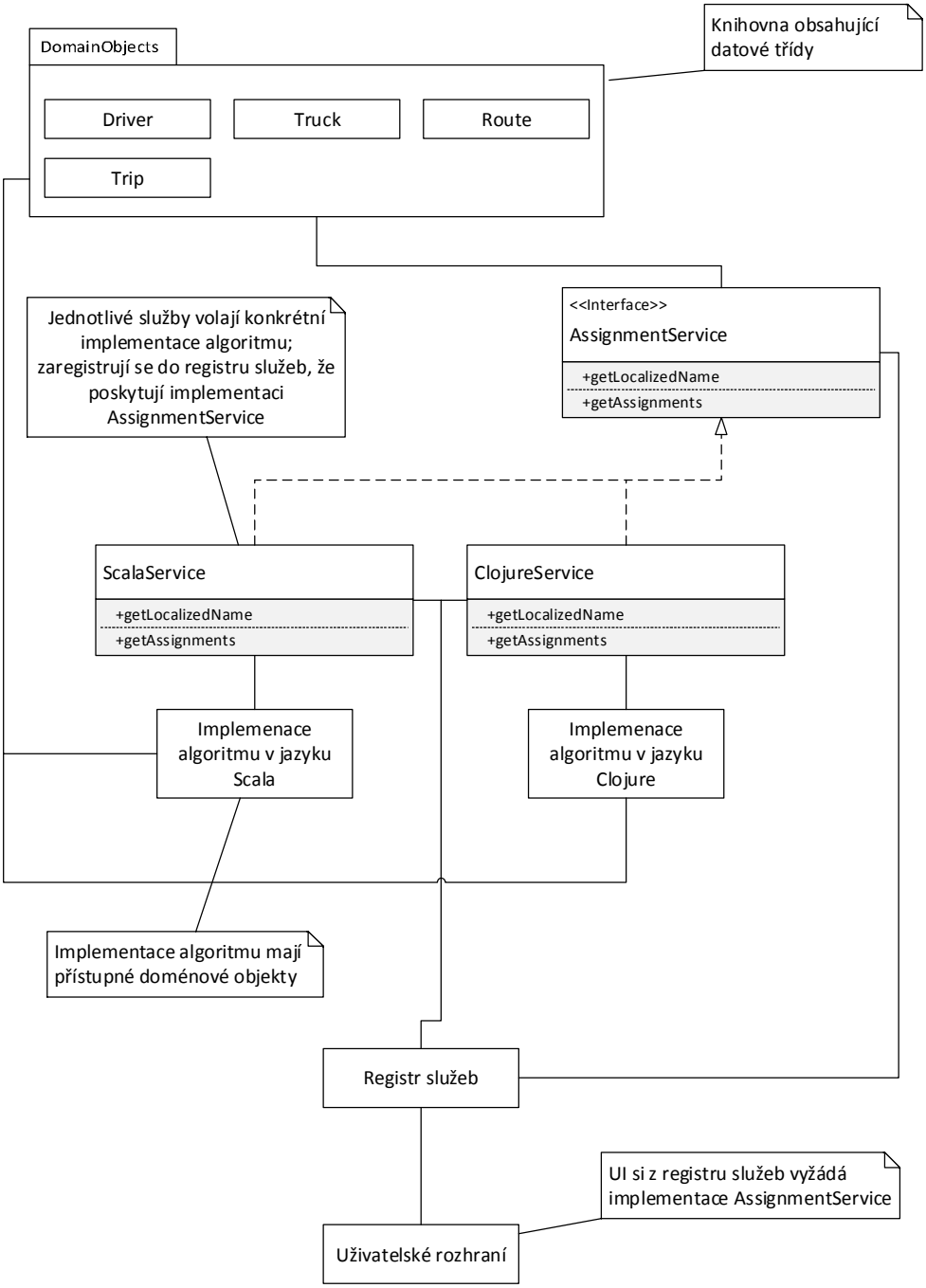
*Ukázka kódu 61 Ukázka převod Java a Clojure kolekcí*

V ukázce kódu 61 lze vidět příklad převodu Clojure vektoru na *java.util.arrayList* a zpět na Clojure vektor. Nejprve je vytvořen vektor obsahující celá čísla 1, 2 a 3. Tento vektor je předán jako parametr konstruktoru třídy ArrayList. Následně pomocí funkce *into* je obsah této kolekce (*java.util.ArrayList* implementuje rozhraní *java.util.Collection*) spojen s prázdným Clojure vektorem (literál „[]“). Výsledek této funkce je opět Clojure vektor obsahující prvky 1, 2 a 3.[21]

[21] Jedná se o tytéž hodnoty v paměti. V době vytváření práce Java Virtual Machine nepodporuje primitivní typy v kolekcích. Proto jsou čísla 1, 2 a 3 zabaleny do třídy Integer, která je referenčním typem a při kopii obsahu vektoru do nové instance třídy ArrayList došlo pouze ke kopii referencí. Stejně potom při kopii obsahu této instance do prázdného Clojure vektoru. Podobným způsobem fungují i převody Scala kolekcí.

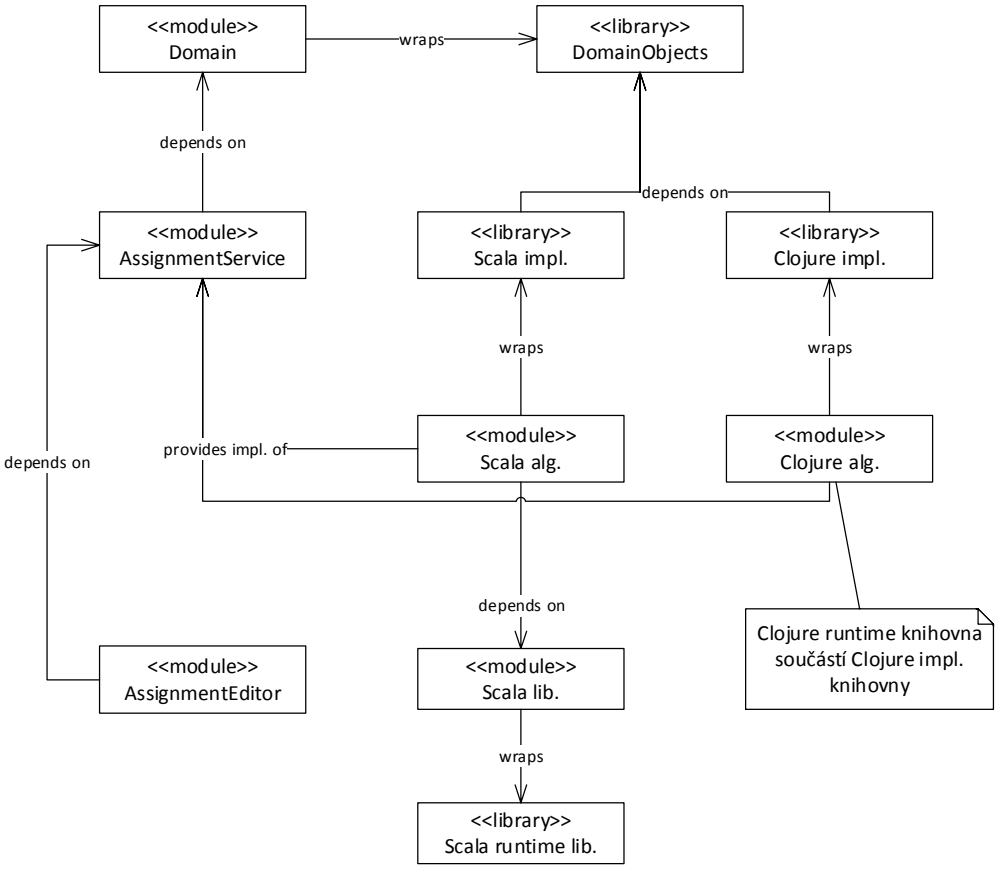
## 1.8. Praktická ukázka

Pro ukázku v této kapitole byla vytvořena jednoduchá aplikace, která slouží k nalezení co nejlepšího rozdělení řidičů, vozidel a tras. Uživatelské rozhraní aplikace je vytvořeno v programovacím jazyku Java (s využitím Netbeans Platform), algoritmus hledající vhodné rozdělení je naimplementován dvakrát, jednou v jazyku Scala a podruhé v jazyku Clojure. Pro obě implementace byly vytvořeny moduly platformy Netbeans, které k nim zprostředkovávají přístup. Uživatelské rozhraní k implementacím nepřistupuje přímo, ale přes registr služeb platformy Netbeans.



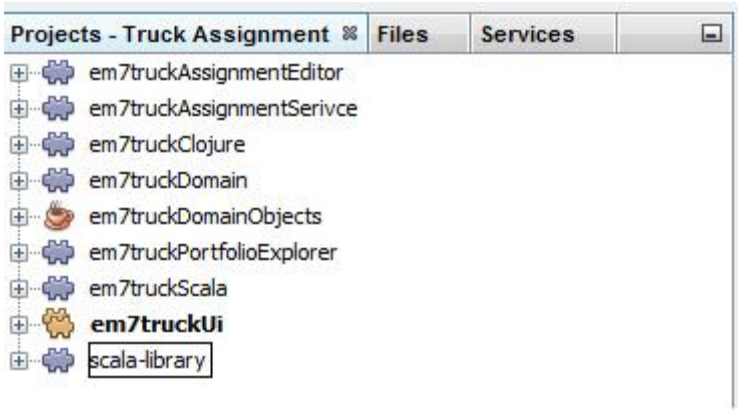

*Obrázek 2 Praktická aplikace: Diagram integrace altoritmů*

Moduly, které zprostředkovávají aplikaci přístup k implementacím přiřazovacího algoritmu, poskytují implementaci rozhraní *AssignmentService* a jsou zaregistrovány v registru služeb platformy Netbeans. Uživatelské rozhraní aplikace si následně může vyžádat všechny služby poskytující implementace tohoto rozhraní. Tímto je zajištěno, že mezi implementací algoritmu a uživatelským rozhraním neexistuje pevná vazba. Běhová prostředí jazyků Scala a Clojure lze buď zabalit do samostatného modulu, nebo integrovat do stávající knihovny/modulu. V případě jazyka Scala byla pro demonstraci zvolena první varianta a v případě jazyka Clojure byla zvolena varianta druhá.




*Obrázek 3 Praktická aplikace: Diagram vztahu knihoven a modulů*

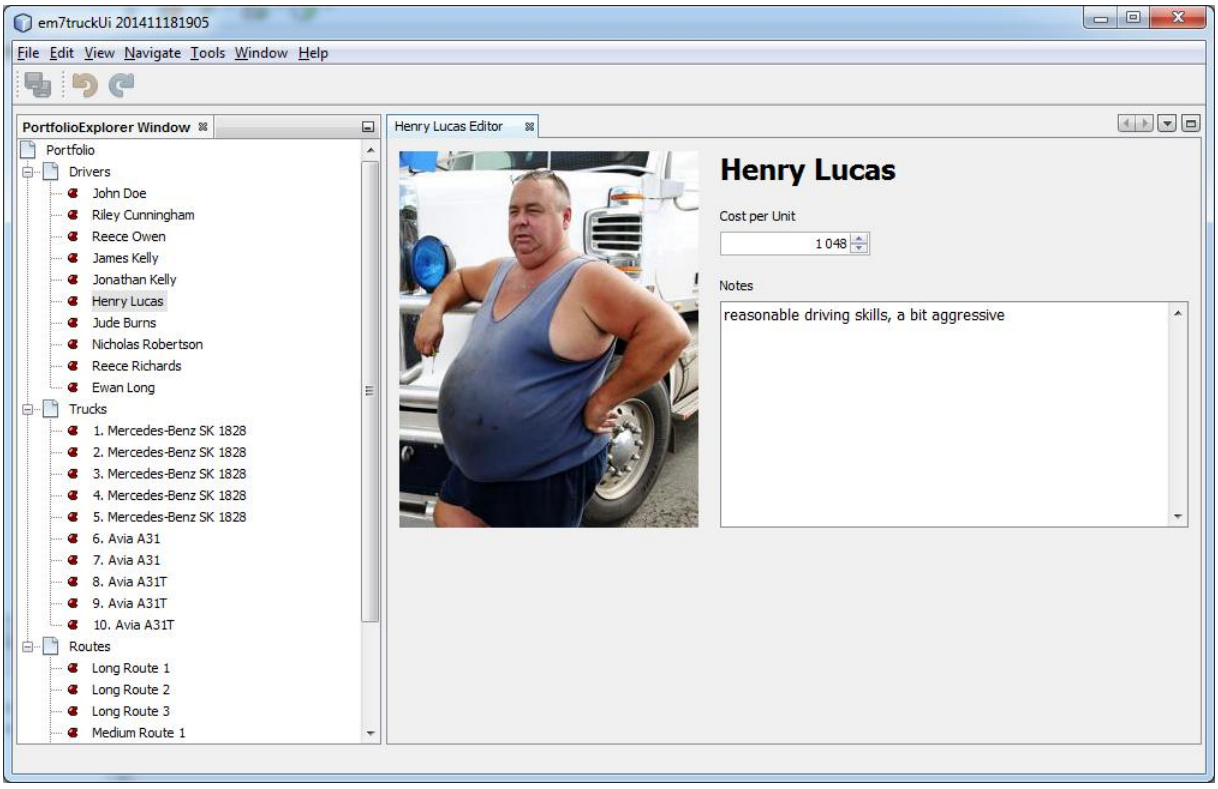
Další součásti aplikace, jako uživatelské rozhraní pro správu řidičů, vozidel a tras či editor přiřazení, byly rovněž vytvořeny jako samostatné moduly.



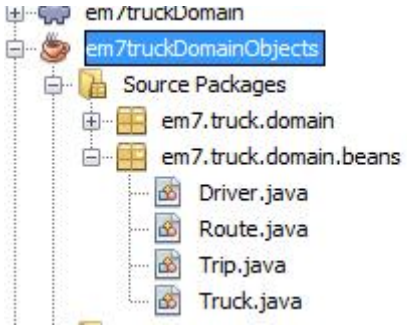

*Obrázek 4 Praktická aplikace: Seznam modulů ukázkové aplikace (vývojové prostředí NetBeans)*

### 1.8.1. Uživatelské rozhraní

Modul *em7truckPortfolioExplorer* umožňuje spravovat informace o řidičích, vozidlech a trasách. Tyto informace slouží jako vstupní data rozdělovacího algoritmu a datové třídy, které je reprezentují, musí být tedy přístupné i pro algoritmy. Proto byla vytvořena knihovna *em7truckDomainObjects*, která tyto datové třídy obsahuje, viz obrázky 4 a 6.



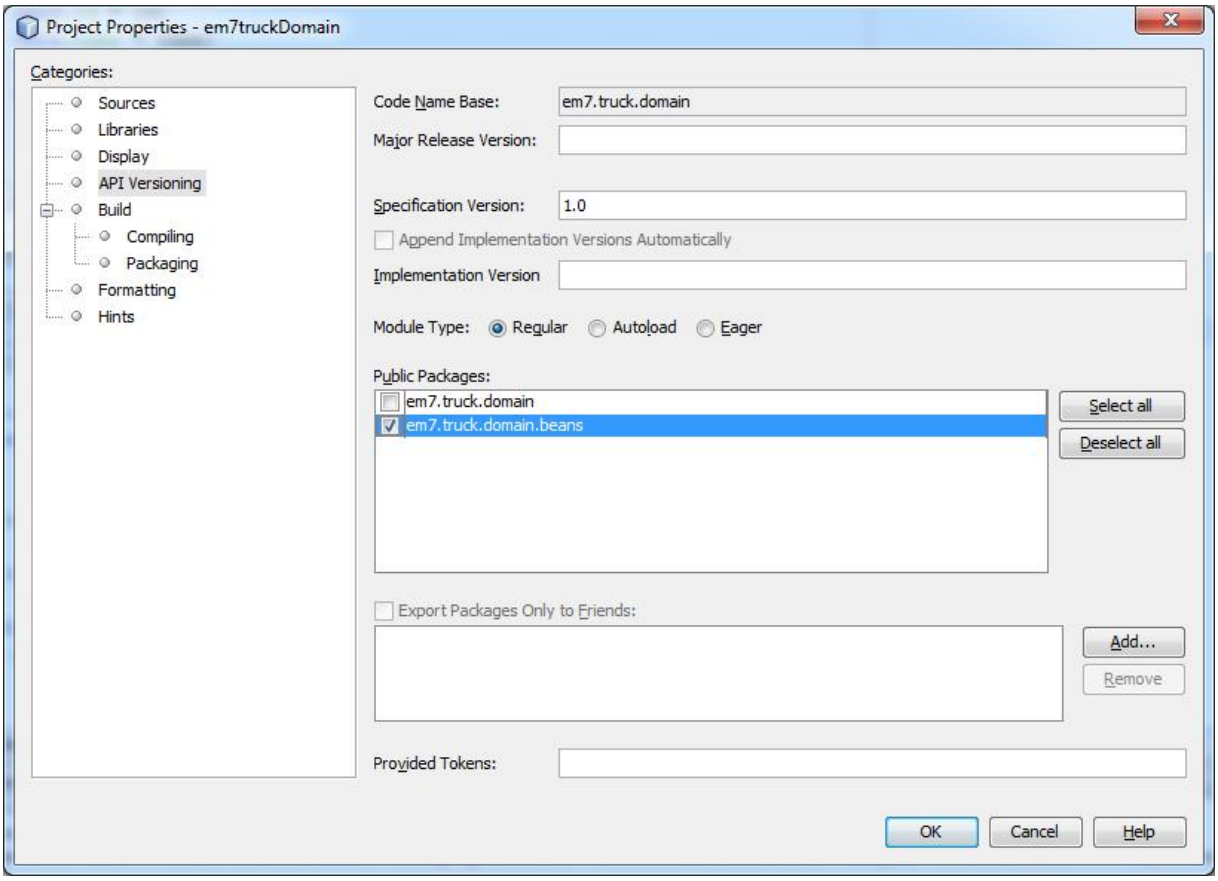

*Obrázek 5 Praktická aplikace: Prohlížení a úprava vstupních informací*




*Obrázek 6 Praktická aplikace: Knihovna s datovými třídami (vývojové prostředí NetBeans)*

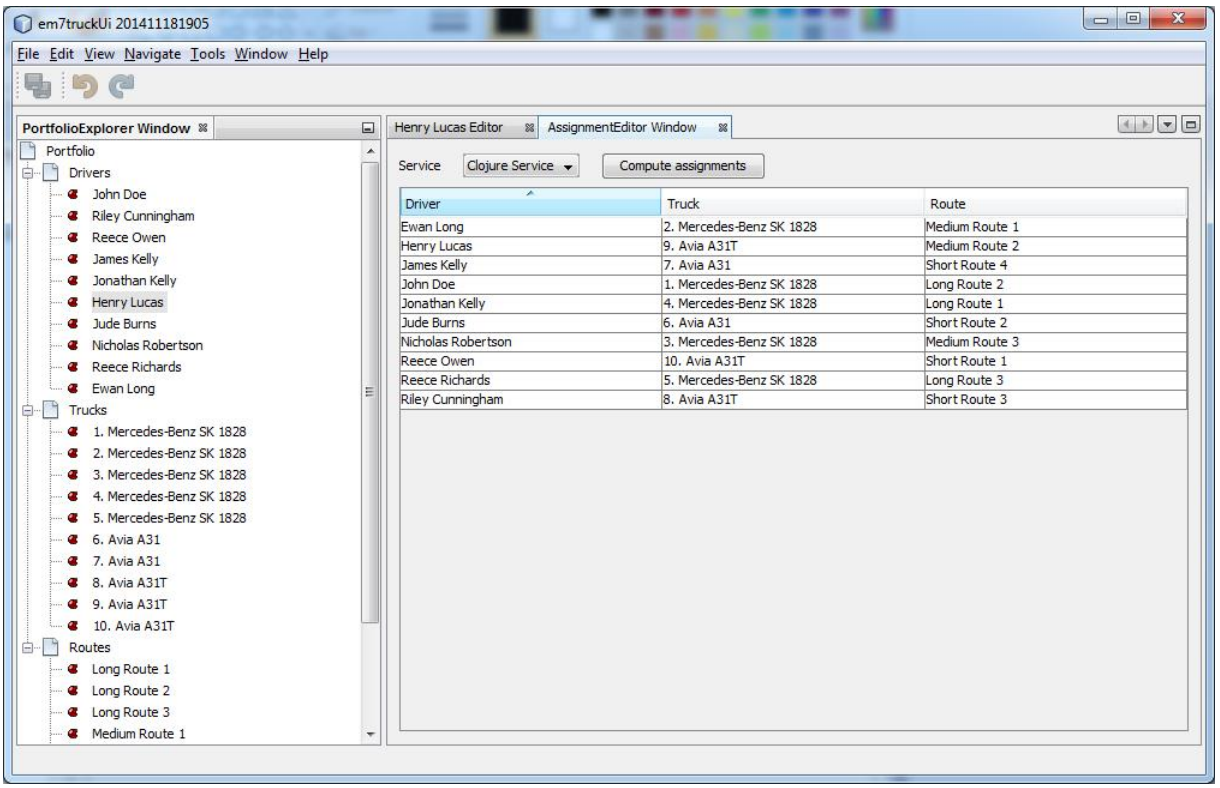
Třídy *Driver, Route, Tirp* a *Truck* jsou pouze datové třídy (s *java.beans.PropertyChangeSupport*).

Knihovna *em7truckDomainObjects* je obyčejný (nespustitelný) jar balíček, nicméně aby třídy, které obsahuje, mohly být využity z více modulů Netbeans Platform, je třeba i pro tuto knihovnu vytvořit modul. Knihovnu s doménovými objekty je třeba do tohoto modulu přidat jako závislost a nastavit balíček s datovými třídami jako veřejný. Tím je zajištěno, že ostatní Netbeans moduly (které deklarují závislot na tomto modulu) mohou využít třídy z tohoto balíčku. S tímto konceptem, kdy některé balíčky modulu jsou veřejně přístupné a jiné skryty, je možno se setkat i v jiných modulárních systémech, např OSGi.




*Obrázek 7 Praktická aplikace: Veřejný balíček modulu (vývojové prostředí NetBeans)*

Modul *em7truckAssignmentEditor* umožňuje spustit přiřazovací algoritmus a zobrazit jeho výsledek. Tento modul nemá žádnou přímou referenci na konkrétní implementace algoritmů, vše je řešeno přes registr služeb platformy Netbeans.




*Obrázek 8 Praktická aplikace: Zobrazení výsledného přiřazení*

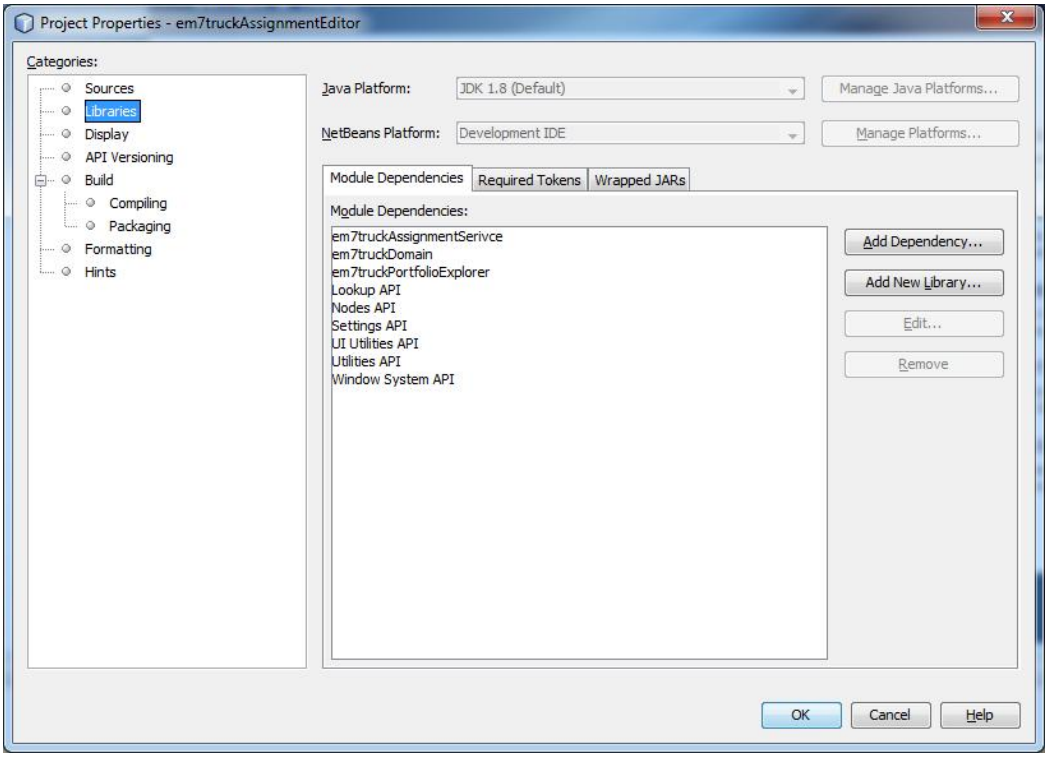
Rozhraní přiřazovací služby je definováno v modulu *em7truckAssignmentService* a implementace tohoto rozhraní slouží pro integraci implementací přiřazovacího algoritmu do aplikace. Balíček *em7.truck.assignment.service* modulu *em7truckAssignmentService* je veřejný a rozhraní je tedy přístupné ostatním modulům, které na něm deklarují závislost.

```
01    package em7.truck.assignment.serivce;
02
03    import em7.truck.domain.beans.*;
04    import java.util.List;
05
06    /**
07     * Descriptor of service for assigning trucks, routes and drivers
08     */
09    public interface AssignmentService {
10
11        /**
12         * Name which should be displayed to the user. Should be
localized
13         */
14        String getLocalizedName();
15
16        /**
17         * Creates the assignment of drivers to trucks and routes
18         */
19        List<Trip> getAssignments(List<Driver> drivers, List<Truck>
trucks, List<Route> routes);
20    }
```

*Ukázka kódu 62 Praktická aplikace: Rozhraní přiřazovací služby*

Rozhraní služby *AssignmentService* definuje dvě metody, *getLocalizedName* a *getAssignments*, viz ukázka kódu 62. Aby balíček *em7.truck.assignment.service*, kteý obsahuje toto rozhraní, byl přístupný ostatním modulům, je potřeba jej zveřejnit (stejně jako v případě modulu s datovými třídami, viz obrázek 7).

Modul obsahující uživatelské rozhraní pro samotné přiřazení pak může definovat závislost na tomto modulu.




*Obrázek 9 Praktická aplikace: Definice závislosti na modulu (vývojové prostředí NetBeans)*

V uživatelském rozhraní, obsaženém v modulu *em7truckAssignmentEditor*, si lze na registru služeb vyžádat všechny služby implementující rozhraní *AssignmentService*, viz ukázka kódu 63. Zavoláním *Lookup.getDefault().lookupAll(AssignmentService.class)* lze získat kolekci všech služeb implementující toto rozhraní. Pokud jsou v registru nalezeny příslušné služby, jsou uloženy do mapy *assignmentServices* a zároveň jsou jejich jména přidána do prvku *serviceComboBox*, který uživateli umožňuje zvolit konkrétní službu, viz obrázek 08.

```
01        /**
02         * Initializes the assignmentServices variable and adds all
the names to the
03         * serviceComboBox
04         */
05        private void createServicesList() {
06            Collection<? extends AssignmentService> services =
Lookup.getDefault().lookupAll(AssignmentService.class);
07            if (services == null) {
08                assignmentSerivces = new HashMap<>(0);
09                return;
10            }
11            assignmentSerivces = new HashMap<>(services.size());
12            serviceComboBox.removeAllItems();
13
14            for (AssignmentService service : services) {
15                if (service != null) {
```

```
16                      assignmentSerivces.put(service.getLocalizedName(),
service);
17
serviceComboBox.addItem(service.getLocalizedName());
18                  }
19              }
20          }
```

*Ukázka kódu 63 Praktická aplikace: Získání seznamu služeb*

Po kliknutí na tlačítko *Compute assignments* je vybraná služba zavolána a jsou jí předána potřebná data. Hotová přiřazení jsou poté zobrazena, viz ukázka kódu 64. Jelikož výpočet vhodného přiřazení může trvat několik sekund, proběhne výpočet ve vlákně na pozadí, aby nebylo blokováno uživatelské rozhraní. Metody *setUiComputingStarted* a *setUiComputingFinished* z ukázky slouží k příslušnému nastavení uživatelského rozhraní (např. zobrazení animovaného „spinneru“ a zablokování tlačítka *Compute assignments*). V metodě *doInBackground* (proběhne ve vlákně na pozadí) vnořené třídy *SwingWorker* lze tedy vidět získání aktuálních dat z datového zdroje (počítá se tedy s aktuálními daty, ale nikoliv s neuloženými změnami), získání uživatelem vybrané služby a spuštění služby. Následně v metodě *done*, která již proběhne v hlavním vlákně aplikace (EDT) a je tedy z ní možno manipulovat s uživatelským rozhraním, je z výsledných dat vytvořen model pro tabulku a je přiřazen tabulce.

```
01          private void computeAssignments() {
02              setUiComputingStarted();
03
04              SwingWorker<List<Trip>, Void> backgroundWorker = new
SwingWorker<List<Trip>, Void>() {
05
06                  @Override
07                  protected List<Trip> doInBackground() throws Exception {
08                      List<Driver> drivers = DataProvider.getDrivers();
09                      List<Truck> trucks = DataProvider.getTrucks();
10                      List<Route> routes = DataProvider.getRoutes();
11
12                      AssignmentService service = getSelectedAssignmentSvc();
13                      List<Trip> assignments = service.getAssignments(drivers, trucks,
routes);
14                      return assignments;
15                  }
16
17                  @Override
18                  protected void done() {
19                      try {
20                          List<Trip> assignments = get();
21                          AssignmentTableModel model = new
AssignmentTableModel(assignments);
22                          resultsTable.setModel(model);
23                      } catch (InterruptedException ex) {
24                          logger.log(Level.WARNING, "Interrupted during showing the
assignment results", ex);
25                      } catch (ExecutionException ex) {
26                          logger.log(Level.WARNING, "Execution exception during showing
the assignment results", ex);
27                      }
28
29                      setUiComputingFinished();
30                  }
31              };
32
33              backgroundWorker.execute();
34          }
```

*Ukázka kódu 64 Praktická aplikace: Volání přiřazovací služby z UI*

Modul *em7truckScala* obsahuje implementaci služby *AssignmentService*, která spustí implementaci přiřazovacího algoritmu vytvořenou v jazyku Scala. Třída *GeneticAlgorithm* představuje implementaci algoritmu a je vytvořena v jazyku Scala.

Podobně modul *em7truckClojure* obsahuje implementaci *AssignmentService*, která spustí implementaci přiřazovacího algoritmu vytvořenou v jazyku Clojure. Zde se třída, která představuje implementaci algoritmu, jmenuje *GeneticAlgorithmImpl*.

Pro ukázku je, jak již název implementujících tříd napovídá, pro řešení úlohy zvolen jednoduchý genetický algoritmus.

```
01    /**
02     * The assignment service implemented in Scala
03     */
04    @ServiceProvider(service = AssignmentService.class)
05    public class ScalaAssignmentService implements AssignmentService {
06
07        @Override
08        public String getLocalizedName() {
09            return "Scala Service";
10        }
11
12        @Override
13        public List<Trip> getAssignments(List drivers, List trucks,
List routes) {
14            GeneticAlgorithm assignment = new
GeneticAlgorithm(drivers, trucks, routes);
15            List<Trip> trips = assignment.solveAssignmentProblem();
16            return trips;
17        }
18
19    }
```

*Ukázka kódu 65 Praktická aplikace: Implementace AssignmentService, jazyk Scala*

```
01    @ServiceProvider(service = AssignmentService.class)
02    public class ClojureAssignmentService implements AssignmentService
{
03
04        @Override
05        public String getLocalizedName() {
06            return "Clojure Service";
07        }
08
09        @Override
10        public List<Trip> getAssignments(List<Driver> drivers,
List<Truck> trucks, List<Route> routes) {
11            GeneticAlgorithm assignment = new GeneticAlgorithmImpl();
12            List<Trip> solution =
assignment.solveAssignmentProblem(drivers, trucks, routes);
13            return solution;
14        }
15
16    }
```

*Ukázka kódu 66 Praktická aplikace: Implementace AssignmentService, jazykc Clojure*

### 1.8.2. Implementace algoritmu v jazyku Scala

```
01    class GeneticAlgorithm(drivers:java.util.List[Driver],
02                           trucks:java.util.List[Truck],
03                           routes:java.util.List[Route]) {
04
05      //the algorithm settings
06      val settings = new AlgorithmSettings(
07                       populationSize = 10,
08                       survivors = 6,
09                       crossBreedProbability = 80,
10                       mutateProbability = 5,
11                       targetFitnessDelta = 100,
12                       maxGenerations = 10000);
13
14      /**
15       * Generates the solution of the problem.
16       */
17      def solveAssignmentProblem():java.util.List[Trip] = {
18
19        //create initial population
20        val population = new Population(drivers, trucks, routes)
21
22        //create genetics
23        val genetics = new Genetics(population, settings)
24
25        //find a solution
26        val solution = genetics.findSolution()
27
28        //to java list
29        val trips = solution.map { g => {
30          val genotype = g.asInstanceOf[population.Genotype]
31          new Trip(genotype.truck, genotype.driver, genotype.route)
32        }}.asJava
33
34        trips
35      }
36
37    }
```

*Ukázka kódu 67 Praktická aplikace: Ukázka implementace přiřazovacího algoritmu v jazyku Scala*

V ukázce kódu 67 lze vidět implementaci třídy *GeneticAlgorithm* v jazyku Scala. Tato třída přebírá vstupní data v konstruktoru (typu *java.util.list*). Metoda *solveAssignmentProblem* vytvoří instanci třídy *Population*, která obsahuje metody pro práci s konkrétní populací (např. vytvoření individua, implementaci fitness funkce apod.) Dále je vytvořena instance třídy *Genetics*, která

obecně implementuje genetický algoritmus a využívá funkcí z třídy *Population*. Řešení (návratová hodnota funkce *findSolution*) je *scala.collection.immutable.List* typu *Genotype* (vnitřní třída třídy Population). Proto je každý prvek kolekce pomocí funkce *map* převeden na typ *Trip*, který je očekáván na straně Java modulu. Následně je Scala list převeden ja *java.util.List* pomocí funkce *asJava*.

```
01      trait GeneticPopulation {
02        /**
03         * The type of a genotype. Individual is a list of genotypes
04         */
05        type Genotype
06
07        /**
08         * Randomly generates a single individual, representated by a list of genotypes
09         */
10        def generateIndividual():List[this.Genotype]
11        /**
12         * Randomly generates the whole population of certain size
13         */
14        def generatePopulation(size:Int):Seq[List[this.Genotype]]
15        /**
16         * Calculates fitness of certain population. Fitness is represented as a list of
tuples, where
17         * first parameter is individual and second is its fitness
18         */
19        def
populationFittness(population:Seq[List[this.Genotype]]):List[(List[this.Genotype], Long)]
20
21        /**
22         * Creates a new individual by mutating the provided one.
23         */
24        def mutateIndividual(individual:List[this.Genotype]):List[this.Genotype]
25        /**
26         * Creates two new individuals by crossbreeding provided ones.
27         */
28        def crossbreedIndividuals(individual1:List[this.Genotype],
individual2:List[this.Genotype]):(List[this.Genotype], List[this.Genotype])
29      }
```

*Ukázka kódu 68 Praktická ukázka: Trait GeneticPopulation*

V ukázce kódu 68 lze vidět trait *GeneticPopulation*, který slouží jako definice rozhraní pro třídy reprezentující populace v genetickém algoritmu. Na rozdíl od rozhraní v jazyku Java lze pomocí trait vynutit více věcí, např. to, že výsledná třída bude obsahovat definici typu *Genotype*.

Ukázka kódu 69 obsahuje vybrané části z třídy *Population*. Lze si všimnout, že konstruktor třídy *Population* vyžaduje Scala sekvence vstupních dat, nicméně v třídě *GeneticAlgorithm* byl zavolán s parametry typu *java.util.List*. Díky automatickým konverzím zde došlo ke konverzi a v třídě

*Population* lze s těmito kolekcemi pracovat jako se Scala typy. V metodě *generateIndividual* si lze všimnout, že vstupní kolekce dat jsou náhodně promíchány. V prostředí jazyka Java by tato operace mohla být nebezpečná, neboť by se mohlo ztratit původní pořadí prvků v kolekci, která je držena v instanční proměnné třídy. Nicméně zde, díky tomu, že *Seq* je v tomto případě *neměnný* typ sekvence, nedojde k ovlivnění pořadí prvků v původní kolekci – je vytvořena nová (jsou zkopírovány pouze reference na prvky) a reference na ni je uložena do lokální hodnoty. Za zmínku také stojí funkce *zip*, která spojí prvky dvou kolekcí a prvky výsledné kolekce jsou sekvencemi obsahující prvky zdrojových kolekcí. Díky tomu lze zjednodušit kód pro vytvoření genotypu skládající se z náhodné trasy, vozidla a řidiče. Ve funkci *generatePopulation* si lze všimnout, že všechna celá čísla v jazyku scala mají metodu *to*. Díky tomu lze vygenerovat sekvenci čísel z konkrétního čísla do parametru funkce *to*. Funkce *genotypeFitness* je fitness funkce, která ohodnocuje konkrétní genotyp individua. V této konkrétní implementaci genetického algoritmu jsou nižší hodnoty fitness považovány za více životaschopné. V úvahu jsou brány náročnost cesty, náklady na řidiče (zahrnující např. mzdu, rychlost, pravděpodobnost nehody…), provozní náklady na vozidlo a rychlost vozidla. Funkce *individualFitness* používá funkci *foldLeft*, která projde všechny prvky kolekce a aplikuje na ně funkci, která přebírá prozatimní výslednou hodnotu (na počátku 0, dále výsledek předešlých operací) a aktuální prvek. V tomto případě se jedná o součet hodnot všech prvků, tedy součet fitness všech genotypů individua. Ve funkci *populationFitness* si lze povšimnout volání funkce *par*. Ta zapříčiní, že další operace nad kolekcí budou probíhat paralelně. Zde byla tato funkce zavolána spíše pro ukázku, nicméně v případě větších kolekcí a náročnějších operací nad nimi lze takto snadno využít více procesorů a zrychlit program. Poslední funkce v ukázce je *mutateIndividual*, kde lze vidět další zajímavé operace nad kolekcemi. Funkce *take* vytvoří novou kolekci, která je tvořena prvními n prvky zdrojové kolekce. Funkce *drop* je opačná funkce – vytvoří kolekci, která je tvořena n + 1. až posledním prvkem zdrojové kolekce. Funkce *head* vrátí první prvek kolekce, funkce *tail* vrátí kolekci bez prvního prvku. Funkce ++ a :: slouží pro spojení kolekcí - ++ pro spojení dvou kolekcí a :: pro připojení prvku na začátek seznamu (ve významu jednosměrně spojového seznamu).

```
01     class Population(drivers:Seq[Driver], trucks:Seq[Truck],
routes:Seq[Route]) extends GeneticPopulation {
02
03       case class Genotype(val driver:Driver, val truck:Truck, val route:Route)
04
05       private val rand = new Random(System.currentTimeMillis())
06
07       /**
08        * randomly generates the individual from provided sequences
09        */
10       def generateIndividual(drivers:Seq[Driver], trucks:Seq[Truck],
routes:Seq[Route]):List[Genotype] = {
11         rand.nextInt()
12
13         val rDrivers = rand.shuffle(drivers)
14         val rTrucks = rand.shuffle(trucks)
15         val rRoutes = rand.shuffle(routes)
```

```
16
17          val zipped = rDrivers zip rTrucks zip rRoutes
18          val seq = zipped.map {case ((driver, truck),route) => new
Genotype(driver, truck, route)}
19          val individual = seq.toList
20
21          individual
22        }
23
24        /**
25         * randomly generates the individual from sequences provided in the
constructor
26         */
27        def generateIndividual():List[Genotype] =
generateIndividual(this.drivers, this.trucks, this.routes)
28
29        /**
30         * randomly generates the whole population
31         */
32        def generatePopulation(size:Int) = {
33          (1 to size).map(i => generateIndividual())
34        }
35
36        /**
37         * computes the fitness of single genotype. Less is better
38         */
39        def genotypeFittness(gen:Genotype) = {
40          val units = gen.route.getUnits
41          val driver = units * gen.driver.getCostPerUnit
42          val truck = units * gen.truck.getCostPerUnit * gen.truck.getTimeFactor
43          driver + truck
44        }
45
46        /**
47         * computes the fitness of one individual. Less is better.
48         */
49        def individualFittness(individual:Seq[Genotype]) = {
50           individual.map { genotypeFittness }.foldLeft(0L)((a, b) => a + b)
51        }
52
53        /**
54         * computes the fittness of whole population
55         */
56        def populationFittness(population:Seq[List[Genotype]]) = {
57          population.par.map { i => (i, individualFittness(i)) }.toList
58        }
```

```
59
60        /**
61         * randomly changes a gene in a genotype of individual
62         */
63        def mutateIndividual(individual:List[Genotype]) = {
64          //gene number to change route with next one
65          val geneToChangeN = rand.nextInt(individual.size - 1)
66
67          //genes which should be persisted
68          val genesBefore = individual.take(geneToChangeN)
69          val genesAfter = individual.drop(geneToChangeN + 2)
70
71          //genes to be changed
72          val genesToChange = individual.drop(geneToChangeN)
73          val firstGene = genesToChange.head
74          val secondGene = genesToChange.tail.head
75
76          //change
77          val firstGeneChanged = Genotype(firstGene.driver, firstGene.truck,
secondGene.route)
78          val secondGeneChanged = Genotype(secondGene.driver, secondGene.truck,
firstGene.route)
79
80          //compose it back to an individual
81          val mutant:List[Genotype] = genesBefore ++ (firstGeneChanged ::
secondGeneChanged :: genesAfter)
82          mutant
83        }
```

*Ukázka kódu 69 Praktická ukázka: Vybrané metody z třídy Population v jazyku Scala*

```
01    class Genetics(val population:GeneticPopulation,
settings:AlgorithmSettings) {
02
03      private val rand = new Random(System.currentTimeMillis())
04
05      /**
06       * Finds the solution using a genetic algorithm according to
settings, using population
07       */
08      def findSolution():List[population.Genotype] = {
09        val firstGen =
population.generatePopulation(settings.populationSize)
10        val solution = _findSolution(firstGen, Long.MaxValue,
Long.MaxValue, 0)
11
12        solution
13      }
14
15      /**
16       * Recursively does one step of genetic algorithm. If previous
and current best fitness is within range in settings,
17       * returns the solution
18       */
19      @tailrec
20      private def
_findSolution(nextPopulation:Seq[List[population.Genotype]],
21          lastBestFitness:Long,
22          prevLastBestFitness:Long,
23          generation:Long):List[population.Genotype]= {
24        val populationFitness =
population.populationFittness(nextPopulation)
25        val fitnessSorted = populationFitness.toList.sortBy(i => i._2)
26        val bestFitness = fitnessSorted.head._2
27
28        if (generation == settings.maxGenerations ||
29            (settings.targetFitnessDelta > 0
30                && (math.abs(prevLastBestFitness - bestFitness) <=
settings.targetFitnessDelta))) {
31          fitnessSorted.head._1 //return the best individual from this
generation
32        } else {
33          _findSolution(createNextGeneration(fitnessSorted),
bestFitness, lastBestFitness, generation + 1)
34        }
35
36      }
```

```
37
38      /**
39       * Creates a new generation for the next step of algorithm
40       */
41      private def
createNextGeneration(currentGenSorted:List[(List[population.Genotype],
Long)]):Seq[List[population.Genotype]] = {
42        //crossbreed individuals
43        val elite = currentGenSorted.head._1
44        val crosBreedIndiv =
rand.shuffle(currentGenSorted.take(settings.survivors))
45        val eliminated = currentGenSorted.drop(settings.survivors + 1)
46
47        //parents
48        val left = crosBreedIndiv.take(settings.survivors / 2)
49        val right = crosBreedIndiv.drop(settings.survivors / 2)
50
51        //children
52        val children = left.zip(right).map(parents => {
53          if (rand.nextInt(100) < settings.crossBreedProbability) {
54            population.crossbreedIndividuals(parents._1._1,
parents._2._1)
55          } else {
56            (parents._1._1, parents._2._1)
57          }
58        }).flatMap(c => List(c._1, c._2))
59
60        //mutate
61        val childrenMutated = children.map { c => {
62          if (rand.nextInt(100) < settings.mutateProbability) {
63            population.mutateIndividual(c)
64          } else {
65            c
66          }}}
67
68        //create next generation
69        val nextGeneration = elite :: childrenMutated ++
population.generatePopulation(settings.populationSize -
settings.survivors)
70        nextGeneration
71      }
```

*Ukázka kódu 70 Praktická aplikace: Implementace genetického algoritmu v jazyku Scala*

V ukázce kódu 70 lze vidět implementaci genetického algoritmu v jazyku Scala. V této implementaci se nepoužívají cykly a proměnné, pouze rekurze a neměnné hodnoty. Třída s metodami pro práci s populací a nastavení genetického algoritmu přichází v konstruktoru třídy a

hlavní funkcí této třídy je funkce *findSolution*, kerá je určena pro volání z jiných tříd. Tato funkce spustí samotný výpočet a vrátí výsledek.

Privátní funkce s názvem *_findSolution* je označena atributem *@tailrec*. Překladač jazyka Scala se snaží rekurzivní funkce optimalizovat, aby nedocházelo k přetečení zásobníku[22]. Nicméně není zaručeno, že optimalizace bude úspěšná, nejedná-li se o koncově rekurzivní funkci. Atribut *@tailrec* zajistí, že v případě neúspěšnosti optimalizace dojde k chybě při překladu, což se při nepřítomnosti tohoto atributu neděje [15]. Tato funkce koncově rekurzivní je, proto optimalizace bude úspěšná. V této funkci se nejprve spočítá fitness každého individua dané generace a je určen nejlepší jedinec. Je-li rozdíl fitness nejlepšího jedince současné a předchozí generace v přednastaveném rozsahu či je překročen maximální počet generací, je tento jedinec označen za řešení úlohy a funkce končí. V opačném případě se vytvoří nová generace a funkce se zavolá znovu.

O vytvoření nové generace se stará funkce *createNextGeneration*. Ta rozdělí současnou generaci na 3 části – elitní jedince, jedince pro reprodukci a eliminované jedince. Elitní jedinci přežijí do další generace, jedinci pro reprodukci se mohou reprodukovat (s určitou přednastavenou vysokou pravděpodobností), přeživší jedinci mohou zmutovat (přednastavena nízká pravděpodobnost) a eliminovaní jedinci jsou nahrazeni novými náhodně vygenerovanými. Pro konkrétní práci s jedinci a genotypy jsou zde využívány metody z trait *GeneticPopulation*; jedná se o obecnou implementaci a je tedy možno tuto implementaci rovnou použít pro řešení jiné úlohy.

Jak je také možno si všimnout z ukázek, v jazyku Scala, ač je staticky silně typovaný, většinou není potřeba uvádět typ hodnot, proměnných či návratových typů funkce, což většinou vede ke kratšímu a přehlednějšímu kódu. Tohoto je možno dosáhnout díky využití typové inference. Pokročilejší vývojová prostředí (např. IntelliJ IDEA se Scala pluginem či Scala-IDE založená na Eclipse) umí typy odvozovat již při psaní kódu a doplňování kódu ve většině případů funguje dobře.

### 1.8.3. Implementace algoritmu v jazyku Clojure

Zatímco vývojová prostředí pro jazyk Scala se, co se týče správy projektů, nijak zásadně neliší od vývojových prostředí pro jazyk Java (v době psaní této práce se téměř vždy jednalo o plug-in do stávajícího vývojového prostředí pro jazyk Java), v případě jazyka Clojure je situace jiná. Z online průzkumu mezi Clojure programátory vyplývá, že nejrozšířenějším vývojovým prostředím pro vytváření Clojure programů je Emacs [22][23]. Pro správu Clojure projektů je pak nejrozšířenější systém Leiningen[24].

Systém Leiningen umí spravovat závislosti a automaticky je stahovat z Maven repozitářů. Vzhledem k tomu, že kód algoritmu závisí na balíčku *em7truckDomainObjects*, je třeba jej přidat do Maven repozitáře. Pro ukázku bude využit lokální Maven repozitář. Nicméně v případě, že by

[22] Java Virtual Machine toto nepodporuje, proto je tato optimalizace implementována v překladači jazyka Scala
[23] Tento editor byl také využit při vytváření příkladů v jazyku Clojure pro tuto práci.
[24] Více informací o tomto systému lze získat z domovských stránek projektu – leningen.org .

na projektu kolaborovalo více lidí, je možno vytvořit nový lokální maven repozitář v adresářové struktuře projektu a tento pak zahrnout do verzovacího systému, který projekt používá[25].

```
01    mvn install:install-file -DgroupId=em7.truck -DartifactId=domainObjects \
02    -Dversion=1.0.0 -Dpackaging=jar -Dfile=em7truckDomainObjects.jar
```

*Ukázka kódu 71 Praktická aplikace: Přidání jar archivu s doménovými objekty do lokálního Maven repozitáře*

```
01    (defproject em7.truck.clojure-assign "0.1.0-SNAPSHOT"
02      :dependencies [[org.clojure/clojure "1.6.0"]
03                     [em7.truck/domainObjects "1.0.0"]]
04      :uberjar-exclusions [#"em7/truck/domain/.*"]
05      :source-paths ["src/main/clojure"]
06      :java-source-paths ["src/main/java"] ; Java source is stored separately.
07      :test-paths ["src/test/clojure"]
08      :resource-paths ["src/main/resources"]
09      :profiles {:doc {:plugins [[codox "0.8.10"]] } ;documentation generator
10                 :uberjar {:aot [em7.truck.clojure.genetic-algorithm-impl]}})
```

*Ukázka kódu 72 Praktická aplikace: Projektový soubor Clojure implementace*

V ukázkách výše lze vidět přidání archivu s doménovými objekty do lokálního Maven repozitáře a projektový soubor pro implementaci přiřazovacího algoritmu v jazyku Clojure. Projektový soubor je v tomto případě běžným zdrojovým souborem jazyka Clojure a lze jej využít. Tato koncepce je velmi rozdílná např. od systémů Ant a Maven, kde projektový soubor je typu XML. U systému Leiningen začíná definice projeků pomocí *defproject*, následuje název a verze a dále různá nepovinná klíčová slova. Hodnoty u klíčových slov jsou většinou vektory (někdy, např. v případě definice závislostí, i vícerozměrné) a mapy jazyka Clojure.

Pod klíčem *:dependencies* jsou definovány závislosti na běhovém prostředí jazyka Clojure a na archivu s doménovými objekty. V tomto případě je žádoucí, aby výsledný jar balíček obsahoval běhové prostředí jazyka Clojure, ale již neobsahoval doménové objekty. Klíč *:uberjar-exclusions* zajišťuje, že soubory, které by se ve výsledném jar archivu nacházely v cestě „em7/truck/domain“ budou vynechány – což jsou právě třídy s doménovými objekty. Znak „#“ před řetězcem uvozuje v jazyku Clojure regulární výraz.

Za zmínku stojí také profil *:uberjar*, tedy profil s nastaveními pro generování „tlustého“ jar archivu. Klíč *:aot* je zkratkou „ahead-of-time“ a jmenné prostory nacházející se ve vektoru pod tímto klíčem budou předkompilovány. Takový krok je v tomto případě důležitý, neboť bude potřeba vygenerovat třídy, které budou volány z jazyka Java.

[25] Vytvoření lokálního Maven repozitáře ve stromové struktuře projektu lze dosáhnout tak, že při spuštění příkazu *mvn deploy:deploy-file* je předán parametr *-Durl=file:local_mvn_repo_dir*. V projektovém souboru systému Leiningen pak lze repozitář přidat pomocí řádku *:repositories {"project" "file:local_mvn_repo_dir"}*.

```
01    public interface GeneticAlgorithm {
02
03          /**
04           * Assigns all drivers a truck and a route, using genetic
algorithm.
05           */
06          List<Trip> solveAssignmentProblem(List<Driver> drivers,
List<Truck> trucks, List<Route> routes);
07    }
```

*Ukázka kódu 73 Praktická aplikace: Rozhraní GeneticAlgorithm pro implementaci v jazyku Clojure*

```
01    (ns em7.truck.clojure.genetic-algorithm-impl
02      "Implementation of the genetic algorithm to be called from
external
03       system."
04      (:require [em7.truck.clojure.population :as population]
05                [em7.truck.clojure.algorithm  :as algorithm])
06      (:import  [em7.truck.domain.beans Trip])
07      (:gen-class
08       :name "em7.truck.clojure.GeneticAlgorithmImpl"
09       :implements [em7.truck.clojure.java.GeneticAlgorithm]
10       :constructors {[] []}
11       :main false
12       ))
13
14    (defn solution->trips
15      "Converts the clojure representation of the solution to list of
16      trips which can be used from java.
17      solution should be a seq of maps"
18      [solution]
19      (let [map->trip #(Trip. (:truck %) (:driver %) (:route %))]
20        (doall (map map->trip solution))))
21
22    (defn -solveAssignmentProblem
23      [this drivers trucks routes]
24      (let [population-fns     (population/create-algorithm-functions
25                                  drivers trucks routes)
26            algorithm-settings population/algorithm-settings
27            solution           (algorithm/find-solution
28                                  population-fns algorithm-settings)
29            trips              (solution->trips solution)]
30        trips))
```

*Ukázka kódu 74 Praktická aplikace: Implementace třídy GeneticAlgorithm v jazyku Clojure*

Pro implementaci genetického algoritmu v jazyku Clojure bylo nejprve vytvořeno rozhraní. Toto rozhraní bylo vytvořeno v jazyku Java a zdrojový soubor je součástí Clojure projektu, viz ukázky výše. Na řádku 08 ukázky 74 si lze všimnout nastavení názvu třídy, která bude vygenerována ze

současného jmenného prostoru. Na dalších řádcích je také vidět, které rozhraní implementuje, že má bezparametrický konstruktor a nemá *main* metodu. Standardně funkce, které začínají znakem „-“, budou součástí třídy (v makru *gen-class* se dá nastavit jiný). Pro tyto metody je vhodné definovat rozhraní v jazyku Java. Pokud by rozhraní definováno nebylo, v jazyku Java by byla tato metoda viditelná s deklarací *public Object solveAssignmentProblem(Object param1, Object param2, Object param3)*. Přestože konkrétnější typy lze definovat i v jazyku Clojure pomocí metadat, znepřehlední se tím kód a není možné specifikovat generické typy[26], což je omezení vyplývající z povahy Java Virtual Machine. Za zmínku také stojí, že nejsou-li metody třídy statické, jejich první parametr je reference na instanci současné třídy. Pokud by třída definovala stav, bylo by možné s ním pomocí tohoto parametru pracovat. Samotná funkce *solveAssignmentProblem* prochází stejnými kroky, jako její ekvivalent v jazyku Scala. Nejprve je vytvořena reprezentace populace, která je předána algoritmu. Výsledné řešení je převedeno na sekvenci instancí třídy Trip. K tomuto účelu je definována privátní funkce *solution->trips*. V této funkci je nejprve definována anonymní funkce, která z HashMapy (mající klíče *:truck*, *:driver* a *:route*) vytvoří příslušnou instanci třídy Trip. Následně je pomocí funkce *map* tato funkce aplikována na všechny prvky vstupní sekvence této funkce. Výsledkem je opět sekvence. Clojure sekvence implementuje rozhraní *java.util.List*, proto je možno ji použít jako návratovou hodnotu jak funkce *solution->trips*, tak i *solveAssignmentProblem*. Lze si všimnout také volání funkce *doall* – tato funkce přebere lazy sekvenci a vyhodnotí ji. Veškeré sekvence jsou v jayzku clojure implicitně lazy, proto je třeba ji před předáním do Java kódu celou vyhodnotit.

V následující ukázce lze vidět vybrané metody ze jmenného prostoru Population, který plní stejnou funkci jako třída Population v případě implementace v jazyku Scala. V Clojure je běžné reprezentovat data pomocí map. Proto v tomto jmenném prostoru není definován typ pro genotyp. Místo toho je zde factory funkce *create-genotype*, která vytvoří mapu obsahující řidiče, vozidlo a trasu. Funkce *create-individual*, na rozdíl od protějšku v jazyku Scala, využívá toho, že funkce pro práci se sekvencemi v jazyku Clojure mohou pracovat s více sekvencemi najednou. Není zde potřeba volat funkci *zip*. Funkce *map* tedy aplikuje předanou funkci na všechny tři sekvence. Výsledkem je sekvence genotypů, která přestavuje individuum. Funkce *create-population* využívá podobného triku jako *generatePopulation* z ukázky v jazyku Scala – na sekvenci čísel od 1 do n se aplikuje funkce. V tomto případě nelze použít zkrácený zápis „#()“, neboť nejsou využity všechny předané parametry. Funkce *genotype-fitness* využívá funkci *bean*. Tato funkce přebírá Java objekt a zpřístupňuje jeho vlastnosti (JavaBean properties) pomocí hashmapy, která je pouze pro čtení. Vlastnosti objektu se pak dají přečíst a zpracovat typickým způsobem pro jazyk Clojure. Nevýhodou je fakt, že se do mapy nedá zapisovat a změnit tak touto cestou vlastnosti původního objektu. Funkce *individual-fitness* využívá funkce *reduce*, která je zde použita ve stejném místě, jako funkce *foldLeft* ve Scala implementaci – tedy pro sečtení hodnot všech prvků v sekvenci.

Funkce *population-fitness* využívá funkce *pmap*, tedy paralelní obdoby funkce *map.* V jazyku Clojure, na rozdíl od jazyka Scala, neexistuje typ pro „paralelní neměnnou kolekci“. Existuje zde relativně malá sada typů a relativně velký počet funkcí, který lze využít s většinou základních typů. Tato vlastnost usnadňuje vytváření nových operací, neboť lze při jejich implementaci využít mnoho již existujících funkcí. Je tím také znesnadněno vytváření nových datových typů, protože již existující funkce v jejich stávajících podobách často využít nelze. Proto je časté reprezentovat

[26] Toto omezení pramení z toho, že generické typy jsou v jazyku Java implementovány na úrovni překladače. Bajtkód Java Virtual Machine s generickými typy nepracuje. [23]

vlastní data pomocí standardních datových typů. Naproti tomu v objektově orientovaných jazycích je běžné definovat relativně velké množství datových typů, ale sada standardních ihned použitelných operací nad nimi je relativně malá a přidání nových není vždy snadné, neboť by musely umět pracovat s velkým množstvím různýcb datových typů.

Další funkcí je *create-algorithm-functions*, která vytvoří anonymní funkce s využitím funkcí definovaných v tomto jmenném prostoru, a vrátí je ve formě mapy. Podobně se chová hodnota *algorithm-settings*, ve které jsou uložena nastavení algoritmu.

```
01    (ns em7.truck.clojure.population
02      (:import [em7.truck.domain.beans Driver Truck Route]))
03
04    (defn create-genotype
05      "Creates a new genotype, which is a map with keys
06      :driver, :truck and :route"
07      [driver truck route]
08      {:driver driver :truck truck :route route})
09
10    (defn create-individual
11      "Creates a random individual from provided list of drivers, truck
12       and routes"
13      [drivers trucks routes]
14      (let [shuf-drivers (shuffle drivers)
15            shuf-trucks  (shuffle trucks)
16            shuf-routes  (shuffle routes)]
17        (map create-genotype
18             shuf-drivers shuf-trucks shuf-routes)))
19
20    (defn create-population
21      "Creates a population of n individuals"
22      [n drivers trucks routes]
23      (map (fn [_] (create-individual drivers trucks routes)) (range 0 n)))
24
25    (defn genotype-fitness
26      "Calculates the fitness of one individual. The individual should
27       be hashmap with :driver :truck and :route from domain beans"
28      [gen]
29      (let [driver (bean (:driver gen))
30            truck  (bean (:truck gen))
31            route  (bean (:route gen))
32            units  (:units route)
33            driv-c (* units (:costPerUnit driver))
34            truc-c (* units (:costPerUnit truck) (:timeFactor truck))
35            total  (+ driv-c truc-c)]
36       total))
37
38    (defn individual-fitness
```

```
39      "Computes the fintess of the individual, which should be
40       a sequence of genotypes."
41      [ind]
42      (->> ind
43           (map genotype-fitness)
44           (reduce +)))
45
46    (defn population-fitness
47      "Computes the fintess of each individual in current population,
48       which should be a collection of individuals. Returns a sequence
49       of maps {fitness individual}"
50      [population]
51      (pmap indiv-in-pop-fit population))
52
53
54    (defn mutate-individual
55      "Randomly mutates a gene in the individual, which is a collection of
genes"
56      [ind]
57      (let [gene-count       (count ind)
58            gene-to-mut-n    (rand gene-count) ;zero based
59            genes-before     (take gene-to-mut-n ind)
60            genes-after      (drop (+ 2 gene-to-mut-n) ind)
61            genes-to-ch      (drop gene-to-mut-n ind)
62            first-gen        (first genes-to-ch)
63            second-gen       (second genes-to-ch)
64            first-changed    (create-genotype (:driver first-gen)
65                                              (:truck  first-gen)
66                                              (:route  second-gen))
67            second-changed   (create-genotype (:driver second-gen)
68                                              (:truck  second-gen)
69                                              (:route  first-gen))
70            mutant           (concat genes-before
71                                     [first-changed]
72                                     [second-changed]
73                                     genes-after)]
74        mutant))
75
76    (defn create-algorithm-functions
77      "Creates a map with functions for algorithm. Input drivers, trucks and
78       routes should be lists of beans.
79       Driver bean should have field costPerUnit (Long).
80       Truck bean should have field costPerUnit (Long) and timeFactor (Int).
81       Route bean should have field units (Long)."
82      [drivers trucks routes]
```

```
83       (let [gen-individual     (fn [] (create-individual drivers trucks
routes))
84             gen-population     (fn [n] (create-population n drivers trucks
routes))
85             population-fitness (fn [pop] (population-fitness pop))
86             mut-individual     (fn [ind] (mutate-individual ind))
87             crossbreed         (fn [ind1 ind2] (crossbreed-individuals ind1
ind2))]
88         {:gen-individual gen-individual
89          :gen-population gen-population
90          :population-fitness population-fitness
91          :mut-individual mut-individual
92          :crossbreed crossbreed}))
93
94     (def algorithm-settings
95       "The map with algorithm settings proper for this population."
96       {:populationSize          10
97        :survivors                4
98        :crossBreedProbability   80
99        :mutateProbability        5
100       :targetFitnessDelta     100
101       :maxGenerations        1000})
```

*Ukázka kódu 75 Praktická aplikace: Vybrané funkce jmenného prostoru Population v jayzku Clojure*

V ukázce kódu níže je vidět obecná implementace genetického algoritmu. Vzhledem k tomu, že jazyk Clojure je dynamicky typovaný a zdrojové kódy většinou nedávají informace o očekávaných typech[27], bývá dobrým zvykem specifikovat očekávané typy v dokumentačních komentářích, především u funkcí a jmenných prostorů. Většina vývojových prostředí je dokáže zobrazit. Stejně tak je možno automaticky vygenerovat API dokumentaci, která je obsahuje. Příklad takového komentáře lze vidět v ukázce níže od řádku 2. V mnoha funkcích jmenného prostoru v ukázce lze vidět využití stejných operací, jako v případě jazyka Scala (např. *take, drop*). Hlavní funkce, *find-solution*, se nachází na konci souboru. Důvod je ten, že překladač jazyka Clojure prochází zdrojový kód pouze 1x a proto je potřeba deklarovat funkce dopředu (např. pomocí *def* či *declare*). Zde je zvolen přístup, že jsou pomocné funkce definovány dopředu. Ve funkci *find-solution* si lze všimnout konstrukce
*(loop (recur))*. Toto je konstrukce, která slouží k vyjádření koncové rekurze v jazyku Clojure, který se takto vypořádává s omezením Java Virtual Machine. Zatímco překladač jazyka Scala dokáže sám identifikovat koncovou rekurzi a provést optimalizaci, překladač jazyka Clojure nikoliv a používá se právě tato konstrukce.

```
01    (ns em7.truck.clojure.algorithm
02      "General implementation of a genetic algorithm. Main function is
03       find-solution which takes the functions needed to run the algorithm
04       and an algorithm settings. Unlike many other genetic algorithm
05       implementations, here the *lower* fitness is better.
06
07       Usage: Call function find-solution, providing functions for working
08              with population and algorithm settings.
09
10    Functions for find-solution is a map:
11       :gen-individual     []          generates the single individual
12       :gen-population     [n]         generates the population of n members,
13                                       population is a seq of individuals
14       :population-fitness [pop]       calculates the fitness for population
15                                       (seq of inidividuals)
16                                       output is a seq of maps
17                                       with keys :fitness :individual
18       :mut-individual     [ind]       mutates the individual, returns new one
19       :crossbreed         [ind1 ind2] crossbreeds two individuals, returns
20                                       seq of two new individuals
21
22    Settings for find-solution is a map:
23       :populationSize        n (Int)   size of population in each generation
24       :survivors             n (Int)   number of survivors from each
generation
25       :crossBreedProbability n (Int)   percentage probability that two of
26                                       survivors would crossbreed (otherwise
27                                       they survive to next generation)
```

[27] Existují typové anotace nebo destrukturování typů parametrů funkcí, které tyto informace poskytují, avšak nevynucují.

```
28        :mutateProbability     n (Int)   percentage probability that a survivor
29                                         would mutate
30        :targetFitnessDelta    n (Long)  if fintess of best individuals of
31                                         different generations is within this
32                                         range, the algorithm stops
33        :maxGenerations        n (Long)  max number of generations")
34
35    (defn crossbreed-individuals
36      "Crossbreeds two individuals or lets them to next generation based on
37       :crossBreedProbability from settings. Returns seq of two individuals."
38      [ind1 ind2 fs settings]
39      (let [prob (rand-int 100)]
40        (if (< prob (:crossBreedProbability settings))
41          ((:crossbreed fs) ind1 ind2)
42          [ind1 ind2])))
43
44    (defn make-children
45      "Creates children for new generation, either crossbreeds or leave
parents
46       based on :crossBreedProbability from settings. Returns seq of
individuals."
47      [crossbreeds fs settings]
48      (let [cnt       (count crossbreeds)
49            left-par  (take (/ cnt 2) crossbreeds)
50            right-par (drop (/ cnt 2) crossbreeds)
51            children  (map crossbreed-individuals left-par right-par)
52            mutated   (map (:mut-individual settings) children)]
53        children))
54
55    (defn create-next-gen
56      "Creates a new generation based on current sorted (with fitness),
57       population functions and algorithm settings. For details of population
58       functions and algorithm settings see the namespace doc. Current sorted
59       generation should be a seq of maps with kesy :fitness (Long)
60       and :individual, sorted by :fitness from lowest (best) to highest
(worst)."
61      [current-sorted fs settings]
62      (let [elite           (:individual (first current-sorted))
63            surviv-with-fit (take (:survivors settings) current-sorted)
64            survivors       (map :individual surviv-with-fit)
65            crossbreeds     (shuffle survivors)
66            children        (make-children crossbreeds fs settings)
67            new-count       (- (:populationSize settings) (:survivors
settings) 1)
68            new-inds        ((:gen-population fs) new-count)
69            next-gen        (concat (cons elite survivors) new-inds)]
```

```
70          next-gen))
71
72      (defn has-target?
73        "Determines whether we have the target solution or not based
74         on current fitness delta (if greater than 0) or generation
75         number, depending on settings."
76        [fit-delta gen-num settings]
77        (let [gen-target?      (= gen-num (:maxGenerations settings))
78              fit-delta-apply? (> (:targetFitnessDelta settings) 0)
79              fit-delta?       (<= fit-delta (:targetFitnessDelta settings))]
80          (or gen-target?
81              (and fit-delta-apply? fit-delta?))))
82
83      (defn find-solution
84        "Finds a best solution to the problem, using population functions
85         (in a map with keys :gen-individual, :gen-population, :population-
fitness
86         :mut-individual and :crossbreed) and algorithm settings (a map with
keys
87         :survivors, :targetFitnessDelta, :mutateProbability, :maxGenerations,
88         :crossBreedProbability and :populationSize). For details see
89         the namespace doc.
90         Result is a seq of maps with :driver :truck and :route."
91        [fs settings]
92        (loop [gen           ((:gen-population fs) (:populationSize settings))
93               last-fit      java.lang.Long/MAX_VALUE
94               prev-last-fit java.lang.Long/MAX_VALUE
95               gen-num       0]
96          (let [pop-fit    ((:population-fitness fs) gen)
97                sorted     (sort-by :fitness pop-fit)
98                best       (first sorted)
99                best-ind   (:individual best)
100               best-fit   (:fitness best)
101               fit-delta  (Math/abs (- prev-last-fit best-fit))
102               has-target (has-target? fit-delta gen-num settings)]
103           (if has-target
104             best-ind
105             (recur (create-next-gen sorted fs settings)
106                    best-fit
107                    last-fit
108                    (inc gen-num))))))
```

*Ukázka kódu 76 Praktická aplikace: Implementace genetického algoritmu v jazyku Clojure*

V definici jmenného prostoru *genetic-algorithm-impl* se nachází makro *gen-class*, které zapříčiní, že z tohoto jmenného prostoru se vygeneruje třída použitelná z externích projektů, např. uživatelského rozhraní vytvořeném v jazyku Java. Ostatní jmenné prostory nejsou přímo takto používány externím kódem, proto použití tohoto makra není nutné.

# 5. Závěr

V této práci byly uvedeny vybrané kreacionální a behaviorální návrhové vzory, které mají význam i ve funkcionálním programování. Pro demonstrace byl zvolen objektově orientovaný jazyk Java a funkcionální jazyky Scala a Clojure. Některé návrhové vzory byly zjenodušeny a v případě jiných bylo pro řešení úloh, které se v objektově orientovaném programování často řeší s využitím návrhových vzorů, využito základních prvků zvolených programovacích jazyků. Strukturální vzory nebyly uvedeny vůbec, neboť jsou příliš spjaty s objektově orientovaným programováním a situace, které řeší, se ve funkcionálním programování zpravidla nevyskytují (nebo, v případě jazyků kombinujících prvky objektově orientovaného a funkcionálního programování, se řeší shodně). Popsány byly také některé vzory typické pro funkcionální programování a také způsob, jak jazyky Scala a Clojure přistupují ke zpracování paralelních výpočtů.

Poslední část se týkala způsobů, jak moduly vytvořené v jazyku Java, Scala a Clojure spojit do jednoho programu, neboť existují jak úlohy, které se ve funkcionálním programovacím jazyku řeší snáze než v objektově orientovaném, tak úlohy, na které se funkcionální programovací jazyky hodí méně. Jazyky Scala i Clojure jsou překládány do bajtkódu Java Virtual Machine, jsou pro ně přístupné všechny knihovny vytvořené pro toto prostředí je možno je z nich využít. Bylo také ukázáno, že knihovny v jazycích Scala a Clojure lze využít z programů vytvořených v programovacím jazyku Java. Pro integraci byla jako příklad uvedena aplikace vytvořená s využitím frameworku Netbeans Platform, která využívala implementace výpočetního algoritmu vytvořené v jazycích Scala a Clojure.

# 6. Literatura


[1] GEPNER, Pawel; KOWALIK, Michal Filip. Multi-core processors: New way to achieve high system performance. In: *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*. IEEE, 2006. p. 9-13.

[2] Java Platform Standard Edition 7 Documentation: Interface Runnable. ORACLE. [online]. [cit. 2014-06-03]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html

[3] MSDN Library: Thread Class. MICROSOFT. [online]. [cit. 2014-06-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/system.threading.thread%28v=vs.71%29.aspx

[4] MSDN Library: Task Parallel Library. MICROSOFT. [online]. [cit. 2014-06-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/dd460717%28v=vs.100%29.aspx

[5] MSDN Library: Parallel LINQ (PLINQ). MICROSOFT. [online]. [cit. 2014-06-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/dd460688%28v=vs.100%29.aspx

[6] Java Platform Standard Edition 8 Documentation: Interface Collection<E>. ORACLE. [online]. [cit. 2014-06-03]. Dostupné z: http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#parallelStream--

[7] MSDN Library: Asynchronous Programming with Async and Await. MICROSOFT. [online]. [cit. 2014-06-03]. Dostupné z: http://msdn.microsoft.com/en-us/library/hh191443%28v=vs.110%29.aspx

[8] EMERICK, Chas, Brian CARPER a Christophe GRAND. Clojure programming. 1st ed. Sebastopol: O'Reilly, 2012, xviii, 607 s. ISBN 978-1-449-39470-7.

[9] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. Design Patterns: Elements of Reusable Object-Oriented Software. 25. vydání. Indianapolis, Indiana, Spojené státy americké: Addison-Wesley, 2002. ISBN 0-201-63361-2.

[10] MSDN Library: Events (C# programming guide). MICROSOFT. [online]. [cit. 2014-06-15]. http://msdn.microsoft.com/en-us/library/awbftdfh.aspx

[11] NORVIG, Peter. Design Patterns in Dynamic Programming. In: [online]. 1998-03-17 [cit. 2014-06-15]. Dostupné z: http://norvig.com/design-patterns/

[12] ODERSKY, Martin. What is Scala. [online]. [cit. 2014-06-15]. Dostupné z: http://www.scala-lang.org/what-is-scala.html

[13] ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE. *The Scala programming language* [online]. [cit. 2014-06-15]. Dostupné z: http://www.scala-lang.org/

[14] TIOBE. *TIOBE Index for June 2014* [online]. [cit. 2014-06-15]. Dostupné z: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[15] Scala language API documentation: scala.annotation.@tailrec. [online]. [cit. 2014-10-19]. Dostupné z: http://www.scala-lang.org/api/current/index.html#scala.annotation.tailrec

[16] HEWITT, Carl, Peter BISHOP a Richard STEIGER. A Universal Modular ACTOR Formalism for Artificial Intelligence. V: Proceedings of the 3rd international joint conference on Artificial intelligence 1973, Stanford, Kalifornie, USA. Strana 235-245.

[17] TYPESAFE INC. Akka: Scala Documentation. [online]. [cit. 2014-11-23]. Dostupné z: http://doc.akka.io/docs/akka/2.3.7/scala.html

[18] TYPESAFE INC. Akka: Scala Documentation: Fault Tolerance. [online]. [cit. 2014-11-23]. Dostupné z: http://doc.akka.io/docs/akka/2.3.7/scala/fault-tolerance.html

[19] Knight, Tom. An Architecture for Mostly Functional Languages. Cambridge, Massachusetts, USA, 1986.

[20] HICKEY, Rich. Clojure: Concurrent Programming. [online]. [cit. 2014-11-27]. Dostupné z: http://clojure.org/concurrent_programming

[21] HICKEY, Rich. Clojure: Documentation [online]. [cit. 2014-11-28]. Dostupné z: http://clojure.org/documentation

[22] GEHTLAND, Justin. State of Clojure 2014 Results. [online]. 2014 [cit. 2015-03-22]. Dostupné z: https://cognitect.wufoo.com/reports/state-of-clojure-2014-results/

[23] GOETZ, Brian. Java theory and practice: Generics gotchas. 2005. Dostupné z: http://www.ibm.com/developerworks/library/j-jtp01255/

# 7. Přílohy

Příloha na CD: Ukázky kódu z kapitol 2 a 4.